A Polymorphic Plan for Database Security

Albert Carlson[*], **Robert Carlson**[¶], **Benjamin Williams**[‡], **Sai Ranganath Mikkilineni**[§], and **Mandeep Singh**[‡]

[*] Computer Science Department, National University, San Diego, CA, USA
[†]Computer Science Department, University of Idaho, Moscow, ID, USA
[‡]College of Business, Delaware State University, Dover, DE, USA
[§]Technical & Information Support Company, 1st Special Forces Group (Airborne), US Army
[¶]Computer Science Department, National University, San Diego, CA, USA
Email: [*]acarlson2@nu.edu, [†]will9847@vandals.uidaho.edu, [‡]SMikkilineni@desu.edu, [§]mandeep.singh35.mil@army.mil, [¶]rcarlson.w@gmail.com

***Abstract*—Databases are one of the most common and important applications in use. The data found in these structures requires a special type of security. Large amounts of data make databases susceptible to language attacks due to the data vastly exceeding the unicity distance for the information it contains. Further, the operation of the database makes it difficult to provide effective changes to the encryption of the resident data. Relational database structures are set up in a manner that makes them uniquely amenable to security using polymorphic techniques. Carlson, et al, posited the principles of polymorphic encryption and random number generators and suggested that the techniques are applicable to many forms of encryption and randomness. This paper targets a mutating encryption structure that allows for independently protecting databases at the field level and mutating that protection over time with access and changes to the constituent data.**

I. Introduction

Data of all types has become a high-value target for attackers. The advent of quantum computing in the post quantum environment (PQE) has accelerated that problem because attackers are now storing data for later attacks. This attack, known as the store now, decrypt later (SNDL) technique [1] anticipates that the increased speed of quantum computers (QCs) and the ability of these machines to break public key encryption (PKE) trapdoor algorithms [2] will result in a trove of valuable data presently stored on classical computers and unavailable to attackers at this time.

A. The Need for Database Security

Databases are frequently targeted by attackers. They contain a great deal of valuable information. Unfortunately, high-quality protection for databases is both expensive and hard to find. These databases frequently change. Protecting a constantly changing corpus of data is difficult and managing the security is often left to a simple cipher architecture that is susceptible to basic decryption techniques. As more security is required the time penalties and costs associated with encrypting the data rise precipitously. Security schemes for protection require the user to optimize the protection for one type of attack and tend to sacrifice protection in other areas related to the data. A complete, fast, and strong database security is required.

B. Database Attacks

Relational databases often contain personal identifiable information (PII) and financial data and are key to automated processes and employee workflows. As such, they are mission-critical and sensitive, which makes them valuable targets for cyber crime. High-profile and expensive cases exist in the news for all of the following attacks:

1) Database Ex-filtration - Database ex-filtration [3] occurs when the full database file is moved outside of the host network and then resold, published, or threatened with exposure unless a ransom is paid. This usually is the result

of a phished or stolen credential attack, but can also be the result of poor access control inside an organization or via an authorized user (such as disgruntled exiting employee). Due to the size of most databases, this type of attack takes time and a significant amount of bandwidth and storage. In many cases, the stolen data is still published even after a ransom is paid. This type of attack is easier to detect, but is often still only discovered after the fact. This attack is nearly impossible to remediate as the stolen data can never be recovered or unpublished.

2) SQL Injection - Unauthorized table retrieval or deletion can be achieved through submitting SQL commands into data fields (known as "SQL Injection"[4]), which are subsequently run in the database. By manipulation of internal logic, it is possible to return data outside of the scope of the field to gain schema data from the database and then subsequently gain unauthorized access to sensitive or valuable information such as credit card numbers or user/customer personal information. Alternatively, this attack can be used to delete information from the database (including a full deletion of all internal tables). This attack is mitigated by internal database limitations which prevent running commands from internal data fields. Deleted data can be restored only if adequate backups exist, and stolen data can never be effectively recovered.
3) File encryption/corruption (Ransomware) - Another attack encrypts or corrupts the database, requiring the organization to pay a ransom for software or a key to reverse these changes and restore the database file to usable condition. This attack does not significantly differ from other ransomware attacks [5] against non-database files, as the employ the same encryption and "bit-twiddling" techniques to achieve there goals. However, they may specifically target database files. This style of attack can also be thwarted by a strong backup policy, but may still result in some data loss. Some variations will specifically target backups for this reason.

C. Goals for a Database Security Scheme

Protecting a database requires meeting a number of goals for securing the data. These goals include:

1) Encrypting all fields separately - Each field in a record must have its own cipher and key pair for protecting the data. The cipher/key pairs should also be different for each record, as well as being unique in each table. Because the cipher/key pair is selected "randomly," it is possible, though improbable, that more than one field or record will have the same cipher/key structure. This condition should be temporary and the pairs will diverge quickly with changing content of the database. The probability of any number of $pr_f(m)$ fields having the same pairs is

$$pr_f(m) = \prod_{i=1}^{m} pr(c_i)pr(k_i) \tag{1}$$

where $pr(c_i)$ is the probability of cipher i being selected as part of the cipher/key pair and $pr(k_i)$ is the probability of key i being selected. This allows for a non-uniform selection for both the cipher and the key. As the number of ciphers and keys increases the probability of duplicated cipher/key pairs also decreases. As long as the random number generator (RNG) that selects the cipher/key pair is independent of each other, either in terms of the RNG or its seeding, the duplication of cipher/key pair will diverge.
2) Low overhead and latency - To enable the database to operate in near real-time, the time to change cipher/key pairs and set up the database should be as small as possible. Fast operation is assured by minimizing the latency, and speedy changes to the changing cipher/key pair encryption can be maintained by using different threads, processors, or graphics processing units (GPUs) available in the computer.
3) Encrypting all fields with unrelated cipher/key pairs - There should be no correlation between the selection of both the cipher and the key used for encryption for any two fields in a database. This requires that the selection RNG or function are independent of each other.

Such a collection of functions of RNGs is indicative of the need for a polymorphic RNG structure.

4) Mutating keys using different TTL schedules - In addition to the need for independent cipher/key pairs the time between changing the pairs to achieve mutating matrix composition. The TTL structure must also be composed of independent values selected by different functions or RNGs. Each TTL must be below the effective unicity distance (n_e) for the data stored in the field. These TTL values are decremented when the field is accessed.
5) Uses strong RNGs, preferably polymorphic RNGs (polyRNGs) [6], to maintain TTL and change schedules - Cryptography and protection of information almost invariably depend directly on the quality of the randomness applied in the process. The strongest RNGs are polymorphic RNGs that mutate over time. Even employing a polymorphic architecture the quality of output is enhanced by using the strongest RNG algorithms available in the software. The output of those algorithms is further enhanced by the polymorphic framework. Taken in total, encryption works best when the selection inputs are as randomized as possible, give the longest cycle length. Since encryption must be deterministic, access to these high quality algorithms also increases security.
6) Have a library of strong ciphers - Using a single cipher is a limitation that attackers can exploit. Even if the cipher is the strongest known eventually the cipher will be identified, patterned, and attacked. Worse, over time attacks are developed or computers evolve in such a way that any encryption will be compromised or broken entirely. Having access to, and using, a variety of ciphers helps to keep information secure. Libraries of hard ciphers give the legitimate users a variety of ways to encrypt and allow for polymorphic encryption. The larger the selection, the more secure the transmission because the probability of attack depends on the hacker selecting the correct cipher/key pair for analysis. Further, libraries of functions allow the users to change the libraries in an irregular but frequent time frame to further frustrate attacks. Ciphers that have fallen to attacks and analysis can be changed as needed in a straightforward and simple manner to maintain security.
7) The ability to store the data in encrypted form - Databases need to be able to access an encryption scheme at the field level so that each entry can be independently encrypted from each other and the rest of the table. Encryption will allow the contents of the field to be safe and encrypted in such a manner to reset the entropy inside the field as needed. Encrypting the field rather than the entire table or database increases the difficulty of decryption by multiplying the number of decryption problems the attacker faces.
8) Be applicable in the classic and PQE environments and on those machines - It should make no difference to the security what type of machine is employed in the effort to recover encrypted data. There should be no need to have one encryption scheme to protect the data in the classical computing environment and another to maintain security in the PQE. This will require using ciphers that are effective in both environments and hard to attack in either. Libraries of such ciphers need to be assembled and maintained for use.

II. Background

In an effort to meet the goals stated for database security, it is important to have a background in a number of areas. The first is in Shannon Theory and Information Theory (IT) [7].

A. Shannon Theory

Security involves a great deal of uncertainty. That uncertainty is sometimes desired, as in the case of the encryption effort, and other times it must be eliminated, as in the case of the decryption action. Shannon described the measure of uncertainty, or "surprise" [8] at the revelation of information, and connected information entropy to physics entropy as introduced by Hartley in the early 1920's [9]. Entropy is defined as

$$H(x) = -\sum_{i=1}^{n} pr(x_i) lg(pr(x_i)) \qquad (2)$$

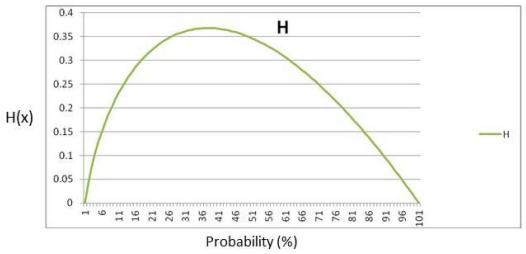

Fig. 1. Shannon Entropy

Entropy has values that run from 0 - .5 for inputs of probabilities corresponding to the range of impossible (0) to must be (1). In both of these cases, there is no uncertainty (surprise) when the input that shows that probability is encountered. This makes sense, as the knowledge about the event or data is known, and revealing another instance of that event adds no new knowledge. Encryption strives for a total ($H(x) = .5$) when encrypting while decryption strives for no uncertainty when working on the decryption of the ciphertext symbol. Therefore, encryption should be a high entropy event.

Solving an encryption problem requires redundancy in the ciphertext. When ciphertext symbols are repeated they reveal information about the plaintext and the encryption that resulted in the observed ciphertext. Redundancy (R_λ) in a language is defined as [8]

$$R_\lambda = 1 - \frac{H(x)}{H_{max}(x)} \tag{3}$$

Redundancy can be viewed as the tendency for a symbol (or block of symbols) to be repeated in a message. In turn, it is the major component of an important measure in encryption known as the "unicity distance."

The unicity distance (n) for a message was defined by Shannon [8] to be the average number of symbols needed to accumulate sufficient information to be able to unambiguously eliminate spurious decryption keys for an encrypted message. It is a measure of the accumulation of information in a message. Unicity distance depends on the size of the alphabet of the language used in the message, the key space of the cipher selected for encryption, and the entropy of the information in the message. The value n is the number of symbols, or blocks, required in order for entropy to be sufficiently reduced to allow decryption. For the message, the unicity distance is given by [8]

$$n = \frac{log(|K|)}{R_\lambda log(|A|)} \tag{4}$$

Carlson showed that the unicity distance can be applied at both the global (message) level and locally for a sub-message, or "shard" [10]. Use of the local unicity distance allowed for applying cipher and key pairs to a sub-message. If the cipher/key pair is changed before the unicity distance is reached the cipher cannot, on average, be broken. Fewer characters mean less data and make decryption impossible. The resulting strategy is to always change cipher/key pairs before the local unicity distance is reached. This allows for the creation of polymorphic ciphers.

B. Polymorphism

Polymorphism is a property that comes from having many forms of the same thing. Implementations of the same quantity are allowed to change at irregular, but frequent, intervals. The main goal of polymorphism is to change an input or control quantity that is used to make something change over time. Polymorphism can be used in encryption and in random number generation, as well as in hiding viruses and in software implementation [11], [12].

Polymorphic ciphers are the more correct name for what were once called "mutating" ciphers or the one time pad (OTP) [8], [10], [13]. These ciphers change their cipher/key pair used to encrypt a shard of data, selecting the different ciphers and/or keys independently of each other. The size of the shard varies between $1 \leq |s| \leq n$ [14]. When $|s| = 1$ the cipher is said to be a Vernam cipher.

Polymorphic ciphers were suggested in the late 1880s by Miller and then rediscovered by Vernam in the mid-1910s. Variations of the cipher were adopted by various countries, including the Soviet Union, after World War II. While Shannon proved that OTPs are the only mathematically provably secure ciphers [8], it is known that if the OTP is not properly implemented with true randomness that it becomes vulnerable to attack[15]. This was demonstrated when the US executed the Venona

attack [16], [17] against the Soviet Union from the late 1930s to the mid-1980s. The crux of the attack was that the Soviets were having issues with the cost of producing true random numbers (TRNGs) and decided to recycle their key pads because of the corpus size of accumulated random numbers. The US was able to determine what keys were used and read messages for decades.

C. Encryption

Vernam based his OTP design on the practice of changing the key for every symbol in a message. However, this is both expensive and slow to implement. Carlson [10] analyzed the system and showed that it is possible to increase the number of symbols encrypted by the same key as long as the unicity distance for the group of symbols does not exceed the unicity distance of that group of symbols. Calling those symbols a "shard," ;this expansion of the use of a single cipher/key pair results in encryption that is both faster and cheaper to implement. Further, if the different shards are not dependent on each other (orthogonal), they can be separated and executed in parallel. This results in faster encryption since the shards can be processed in parallel. Any extra effort in making the shards and operating on them is subsumed in the parallel treatment.

The process of splitting a message into shards is highly dependent on the content of the message. Shards are typically randomly sized to be less than n in length for that shard. There are other ways to make shards than analyzing and splitting up a message. Some data and information come naturally pre-sharded, or more correctly, preprocessed into sufficiently small packets of data for immediate protection using polymorphic encryption. One such application is the database where the data is separated into fields that approximate the sharding process. Fields in a record are typically short and, due to the application, reduce redundancy wherever possible so that they meet the 5th normal form (5NF) [18].

Records and fields that are accessed from databases are typically not repeatedly accessed over and over in a row. That means that the fields normally appear in data stream accesses separated by other record or field accesses. With varying record and field sizes the repetition of data using the same cipher/field pair is irregular and forces the attacker to guess where the same cipher/field pair begin and end. Without the exact boundaries of that data, the corpus used to decrypt the data will be mixed with encryption for other cipher and key pairs. As a result, the data is tainted and the problem is one where bad data is mixed with the correct data for the field(s). Data arranged in this manner allows for redundancy in the fields to be controlled at the data level by using a scheme to change data and add entropy that is linked to each unique field and changing the cipher/keys individually for the various fields. An example of the Time to Live (TTL) is shown in Figure 2.

III. Time to Live (TTL)

In the illustration, there are six different values that need to change in a polymorphic system. Although only one number is labeled cipher and one key, the lines labeled "effect" can also be either a cipher or key. What is shown is how the ciphers or keys change over time and the effect it has on the system state, which can also be considered and treated as a key for the system. The length of time that the value is stable is governed by the entropy and translated into a time, or number of key accesses. In practice, the time that a key is used and the number of accesses are equivalent but expressed in different units. Time is expressed in seconds, and accesses as a unitless number. Each of the selections is run by its own TTL on a different schedule than the other cipher or key selections. They will occasionally change at the same time, but most of the transitions will be unique and independent of each other. Even though the changes happen at what can be considered a relatively slow rate the variation between the different cipher and key selection result in a change of the total encryption state that occurs much more rapidly than any single cipher/key pair. The changes in values are known as "time domain multiplexing," or TDM [19].

In order for an attacker to be able to decrypt data from multiple streams the attacker has to know the cipher and key pair for each stream of data. If each field is a different stream then the attacker must break each data stream cipher/key pair. The combination of ciphers and keys constitutes a "composite"

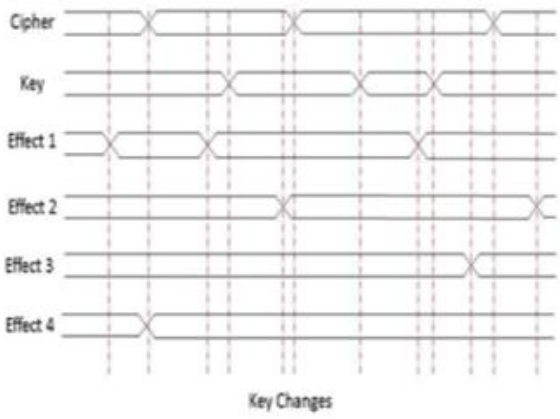

Fig. 2. Time Domain Multiplexing Example

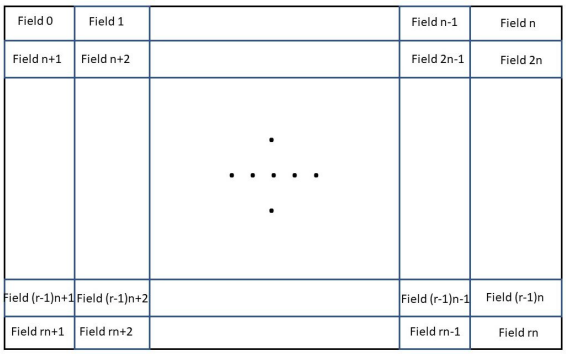

Fig. 3. A Database

key composed of each of the cipher and key pairs. Each time one of them changes the composite key also changes. The result is that the composite key changes very rapidly compared to the individual changes of any single cipher or key. The faster the ciphers and keys change, in general, the more secure the data. Since the pairs are changing at rates faster than the information can accumulate sufficiently to meet the unicity distance, security is maximized for the individual data streams without having the expense of changing every character in every data stream.

IV. Implementation

Implementing the new algorithm requires mirroring the structure of the database. A relational database is composed of one, or more, tables. Each table consists of records that span the row of the table with fields of data comprising the columns. This structure is shown in Figure 3. Ignoring the relative sizes of the data stored in the various fields the table can be simplified into a regular rectangle with a size ($|M|$) given by

$$|M| = |records| \times |fields| = rf \tag{5}$$

Three such structures are required for this algorithm. The first holds the TTL assigned to each field, the second holds the cipher data for encrypting data, and the final table contains decryption mappings. Tables have the structure shown in Figure 4.

In the first table, the TTL for a particular field is saved. TTLs are independent of each other. Each time an access to that field occurs the TTL in the TTL is decremented. When the TTL reaches 0 the field is marked as busy and then reprovisioned. Reprovisioning consists of selecting a new TTL that is shorter than the unicity distance, placing that in the TTL table, selecting a new cipher and key, and then replacing the encryption and decryption data in the appropriate tables.

The encryption table is similar to the TTL table in structure, except that each cell in the table consists of a list of the plaintext alphabet paired with the encrypted data for that entry. An encryption mapping as the formula

$$A \mapsto A' = PT \mapsto CT \tag{6}$$

where PT is the plaintext and CT is the ciphertext, this is equivalent to, and can be implemented with, a Python dictionary or a C++ map data structure. Since the encryption is 1:1 and onto, this also means that it is possible to construct the mapping for

$$CT \mapsto PT \tag{7}$$

Such mappings are possible because all ciphers are, at their heart, substitution (S) ciphers [20]. Carlson, et al [10], [21], [22] proved that unless randomizing functions are inserted and used that all ciphers can be converted into an equivalent S cipher. This includes block ciphers [23] and more complex product ciphers. Therefore, if each cell in the encryption (or decryption) table, the cell can be built to contain that mapping. Such a mapping can be used for a fast encryption (or decryption) action. Assume that the table for encryption is located at memory location Loc. Also, assume that the contents of a memory

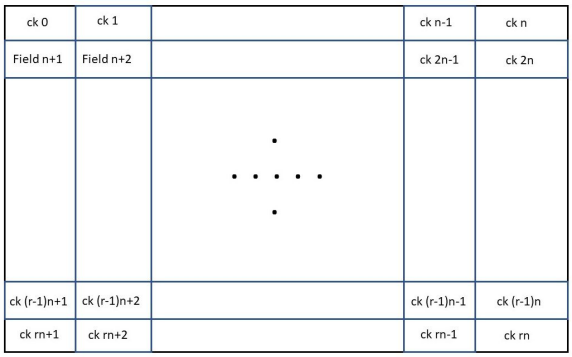

Fig. 4. Cipher and Key Pair Matched to a Database Structure

location are denoted by $[Loc]$. Then the alphabet PT character located at $Loc + PT$ is given by

$$[Loc + PT_i] = E_{c,k}(PT_i) = CT_i \tag{8}$$

In this manner, the location contained by each cell is minimized. Similarly, for a decryption table the roles of CT and PT are reversed. That is, to accomplish a quick decryption, the contents are written as

$$[Loc + CT_i] = D_{c,k}(CT_i) = PT_i \tag{9}$$

These table structures are shown in Figure 4. These two tables can be combined with a small amount of algebra such that

$$CT_i = [Loc + 2(PT_i)] \tag{10}$$

and

$$PT_i = [Loc + 2(CT_i) + 1] \tag{11}$$

If each field has an alphabet of A_j characters, then to operate on cell c, the memory location is given by

$$Loc = 2 \sum_{i=0}^{c-1} |A_{c_{i-1}}| \tag{12}$$

Each time the cell is provisioned, the data is written to these memory locations with the latest cipher/key pair information.

With the data structures defined, the algorithm can then be specified. There are a number of steps in this algorithm. They are

1) Size the database to size the needed control tables - Count the number of records and the number of fields in the record.
2) Set up an RNG for creating the data related to security - Set up an RNG, preferably a polymorphic RNG to select the TTL table and the cipher/key pairs for use in encryption and decryption.
3) Use the RNG to create the TTL matrix for each field in a table - This table will provide the ongoing TTLs for security.
4) Create a cipher/key table using the RNG - Next, create a matrix that is as large as the database tables. Populate the table with the encryption and decryption lists. Share the required data with the user in the most secure manner available. These tables are saved locally so that the user and provider can synchronize at any time after the initial database setup.
5) Initially encrypt the database - Both sides are now available to use the database in safe mode. Data is transferred in encrypted form, although the local database may still have to decrypt the information.
6) For each access to the database, including reading and making inquiries the following steps need to be followed:

    a) Access the data - Data will need to be retrieved from the database and decrypted for handling.
    b) Do the requested action - Do whatever is required to be done in the database. This is the step in which the database operates as normal.
    c) Decrement the TTL for data accessed- If the TTL is greater than 0, do not rewrite the fields. However, if the TTL reaches 0 for any of the fields accessed, mark the record as being repartitioned. Next, generate a new TTL, new key, new cipher, placing them in the correct fields, Then redo the encryption/decryption table for the record, and re-encrypt the data for the field, rewrite it to the field, and mark the record as available. Then continue normal database operation. The database can still be used without time penalty or latency so long as no field marked for repartitioning needs to be accessed.

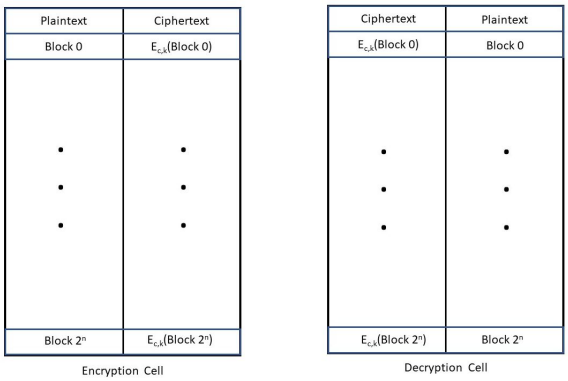

Fig. 5. Encryption and Decryption Cell Structure

If an access does require a locked field, then operations must wait until the field is repartitioned and cleared for further use.

V. Analysis

Breaking up the contents of a table into fields is similar to the sharding process used in polymorphic encryption [14]. The contents of each field are relatively short, with respect to the data found in the entire database. It is unlikely that the field will contain enough data to exceed the effective unicity distance [10] for the cipher and key pair. Relatively small shards ensure the security of the encryption for the field.

Polymorphic encryption avoids exceeding the unicity distance of the ciphers used by splitting the data into small shards, where each shard is provably smaller than the unicity distance of the cipher that will be used to encrypt it. This guarantees that the encrypted message does not contain sufficient information to derive the key used to encrypt any given shard, which means that the encrypted message itself cannot be cracked except by a brute-force attack. Better yet, even a brute force attack cannot be effective, because for a message of any given length, there are many other possible messages that could be discovered that are not the correct one, and the encrypted data provides no clue as to which coherent message is the correct one. For example, if we encrypt the word ”and” with a good encryption algorithm, no patterns will remain that will allow an attacker to know that ”and” is the correct result while ”the”, ”out”, ”off”, ”yam”, ”eel”, or any other three letter word is not correct. Polymorphic encryption breaks down messages into shards to guarantee that this applies to each and every shard independently, and thus also to the entire message as a whole.

With polymorphic encryption, because many subkeys are used, an attack is far more likely to come across a coherent but incorrect message than it is to come across the one correct one, and even if an attacker could instantly test all possible keys, there would be no way to distinguish between the enormous number of coherent but incorrect messages and the one correct one. This is further exacerbated in a database, where fields are not related by grammatical constructs, punctuation, or formatting, which means that the number of possible coherent but incorrect decryptions is far larger. Imagine an encrypted nine-character username. How many other possible values are there that would look like valid usernames but are not? There are more than 20 billion possibilities with just uppercase letters, lowercase letters, and numbers. Even if we remove all of the ones that do not look like something someone would choose as a username, we still have tens or hundreds of thousands, but only one of those is the correct one. Passwords are worse, because they are often longer, less coherent, may contain special characters, and are often hashed before they are stored. Smaller fields, such as numerical values, are even worse still, because every attempted decryption will produce a valid number. An attacker cannot rule out most values simply because they do not look like something a human would choose. If each field is encrypted with its own subkey, the odds of guessing the correct valid decryption for enough fields to compromise anything of value become astronomical. The problem becomes so difficult that the odds of cracking it are no better than the odds of randomly guessing the correct values.

Once a cipher/key pair has been selected, two entries are made in the corresponding blocks for the encryption and decryption matrices. Since all ciphers follow the 1:1 and onto principle and act as substitution mappings [20] and can be simply precomputed. By making a list of the plaintext alphabet and pairing them with their corresponding list of ciphertext in a simple lookup table is possible. For an alphabet of $n = 2^b$ characters, the complexity

[24] of preparing the list for each block for those n symbols is $\Theta(n) = n$. In operation, the encryption and the decryption process both have the same complexity, that of a single lookup. The complexity and latency of a lookup are known to be $\Theta(n) = 1$.

But this does not present the entire story. Although there can be many entries in the table these structures can be large, reprovisioning is an orthogonal problem to accessing the contents of the database. Therefore, in a multi-threaded or multi-core system, the reprovisioning of an encryption/decryption block can be done while the database is in use. Any overhead cost will be subsumed during regular functioning. The probability of any one block being accessed during operation ($pr_a(x)$), assuming random access to the database, is

$$pr_a(x) = \frac{1}{|t|} \tag{13}$$

where $|t| = |rf|$ and $|r|$ is the number of records, each having $|f|$ fields. The exact amount of time used is highly dependent on the number of threads, cores, or general processing units (GPUs). The larger the number of tables and the contents in the tables, the less likely there will be a collision with the reprovisioning process and cause delays in transmission.

From the memory standpoint, evaluating the algorithm is relatively simple. For each field in the table the following items are required:

- TTL - One integer whose size is

$$|TTL| \approx \lceil lg(n) \rceil \tag{14}$$

  which is $\Theta(lg(n))$.
- Encryption block - Requires one entry for each character in the block alphabet. That is a space of $\Theta(|A|)$ is required.
- Decryption block - This memory structure is the same size as that for encryption, since there is a 1:1 correspondence between PT characters and CT characters.

$$\begin{aligned} |memory| &= |TTL| + |encryption| + |decryption| \\ &= |TTL| + 2|encryption| \\ &\approx 2|encryption| \end{aligned} \tag{15}$$

which together is $\Theta(|A|)$ and must be less than the total of

$$|memory| \leq (2+1)|encryption| \leq 3|encryption| \tag{16}$$

where the size of the encryption block ($|blk|$)is

$$|blk| \leq \lceil lg(|A|) \rceil \tag{17}$$

and the total memory required is bounded by

$$|memory_{total}| \leq |blk|memory \leq 3|blk||encryption| \tag{18}$$

VI. Conclusion

It is both possible and fairly easy to protect databases on a field-by-field basis. Further, this protection can be based on using polymorphic principles that mutate the encryption over the fields and treats each field as an independent, orthogonal encryption problem. The independent nature of the interactions uses TDM to create a complex, composite key for the database.

This paper presents a plan, along with an operational algorithm, that implements a polymorphic security system for a database. It is expansible and includes an analysis of the memory and time requirements for the system. The algorithm can be adapted to any database, but was shown for a relational database having any number of tables, records, and fields. Encryption and decryption are done using a fast table implementation, which was also presented. This approach is a new approach for securing data in database form and holds promise for future development and applications. It is equally safe and applicable in the classical and PQE, as long as the contents for the library are properly composed of algorithms that are both strong and quantum resistant. This typically means symmetric algorithms with large key spaces and strong trapdoor mathematics. However, this algorithm allows for customization and is an engine that will keep data safe.

References

[1] Duncan Riley. Half of organizations worry about quantum 'harvest now, decrypt later' attacks.

[2] Johannes A. Buchmann, Evangelos Karatsiolis, and Alexander Wiesmaier. *Introduction to Public Key Infrastructures*. Springer-Verlag, Berlin, Germany, 2013.

[3] Eduard Kovacs. Researchers devise "perfect" data exfiltration technique. *Security Week*. https://www.securityweek.com/researchers-devise-perfect-data-exfiltration-technique/.

[4] Mohd Amin Mohd Yunus, Muhammad Brohan, Nazri Mohd Nawi, Ely Salwana, Nurhakimah Najib, and Chan Liang. Review of sql injection : Problems and prevention. *JOIV : International Journal on Informatics Visualization*, 2:215, 06 2018.

[5] A. Young and M. Yung. Cryptovirology: extortion-based security threats and countermeasures. *IEEE Symposium on Security and Privacy*, page 129 – 140.

[6] Yelai Feng, Huaixi Wang, Chao Chang, Hongyi Lu, Fang Yang, and Chenyang Wang. A novel nonlinear pseudorandom sequence generator for the fractal function. *Fractal and Fractional*, 6:589, 10 2022.

[7] Thomas Cover and Joy Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc, New York, 2nd edition, 2005.

[8] Claude Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656 – 715, 1949.

[9] Paul Garrett. *The Mathematics of Coding Theory*. Pearson/Prentice Hall, Upper Saddle River, 2004.

[10] Albert Carlson. *Set Theoretic Estimation Applied to the Information Content of Ciphers and Decryption*. PhD thesis, University of Idaho, 2012.

[11] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471 – 522, December 1985.

[12] John C. Reynolds. Types, abstractions, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, 1983.

[13] Gilbert Vernam. Secret signal system.

[14] Albert H. Carlson, Indira Kalyan Dutta, Bhaskar Ghosh, and Michael Totaro. Modeling polymorphic ciphers. *FCST 2021: International Conference on Foundations of Computer Science Technology*, 2021.

[15] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. Cryptanalytic attacks on pseudorandom number generators. In *Fast Software Encryption, Fifth International Workshop Proceedings (March 1998)*, pages 168 – 188. Springer-Verlag.

[16] John Earl Haynes and Harvey Klehr. *Venona: Decoding Soviet Espionage in the United States (Yale Nota Bene)*. Yale University Press, 1999.

[17] Albert Carlson, Sai Ranganath Mikkilineni, Michael W. Totaro, and Christopher Briscoe. A venona style attack to determine blocksize, language, and attacking ciphers.

[18] C. J. Date. *Database Design and Relational Theory: Normal Forms and All That Jazz*. Apress, New York, NY, 2nd edition, 2019.

[19] Jim Kurose and Keith Ross. *Computer Networking A Top-Down Approach*. Pearson, 6th edition, 2013.

[20] Horst Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15 – 20, 1973.

[21] Bhaskar Ghosh, Indira Dutta, Shivanjali Khare, Albert Carlson, and Michael Totaro. Isomorphic cipher reduction.

[22] Albert Carlson, Bhaskar Ghosh, Indira Dutta, Shivanjali Khare, and Michael Totaro. Keyspace reduction using isomorphs.

[23] Uli Maurer and James Massey. Cascade ciphers: The importance of being first. *Journal of Cryptology*, 6(1):55 – 61, 1993.

[24] Oded Goldreich. *Foundations of Cryptography I*. Cambridge University Press, Cambridge, 2001.