Master’s Project Report

Tamper evident encryption of integers using keyed Hash Message Authentication Code

Bradley A. Baker
B.S. University of Colorado, 2003

A Project
Submitted to the Graduate School Faculty of the University of Colorado at Colorado Springs
In Partial Fulfillment of the Requirements For the Degree of Master of Engineering in Information Assurance
Department of Computer Science
Fall 2009

November 16, 2009
This project for the Master of Engineering in Information Assurance degree by
Bradley A. Baker
Has been approved for the
Department of Computer Science
By

Dr. C. Edward Chow

Dr. Jugal Kalita

Dr. Terrance Boult
Abstract

The focus of this project is confidentiality and integrity of data in a database environment, particularly numeric data. Databases are used to store a wide variety of sensitive data, including personally identifiable information and financial records. The quantity and value of sensitive data is constantly increasing, and this data must be protected from unauthorized disclosure or modification.

This project aims to provide confidentiality and integrity of data through an encryption scheme based on the keyed Hash Message Authentication Code (HMAC) function [3, 12]. The encryption scheme implemented in this project extends and improves the HMAC based encryption scheme presented in [1]. The result is a symmetric encryption process which can detect unauthorized updates to ciphertext data, verify integrity and provide confidentiality. The encryption scheme is implemented in a database environment and the developed process is named “HMAC based Tamper Evident Encryption,” referred to as HTEE in this paper.

This scheme provides an alternative to standard approaches that offer confidentiality and integrity of data such as combining the Advanced Encryption Standard (AES) algorithm with a hash digest. These standard approaches can be difficult to implement, and may not be ideal for all environments. The purpose of the HTEE scheme is to provide efficient encryption that supports data integrity in a straightforward process, to investigate the use of HMAC for reversible encryption and key transformation, and to improve upon an existing method.

To introduce the design, this encryption scheme processes positive integer values and decomposes them into components, or buckets, using modular arithmetic. The buckets are encrypted using the HMAC-SHA1 function, where the authentication code represents the ciphertext. The secret key used for HMAC is modified for each plaintext value using a key transformation process. Decryption is performed with an exhaustive search for authentication code matches. Unauthorized changes to ciphertext values or related data are detected during the decryption process, when the plaintext result cannot be found. The design of bucket decomposition makes the exhaustive search process feasible for large numbers. The key transformation process supports tamper detection and improves security.
# Contents

1. Introduction......................................................................................................................1
   1.1. Motivation .........................................................................................................................1
   1.2. Problem Summary ...........................................................................................................2
   1.3. Project Overview .............................................................................................................3
   1.4. Encryption Example ........................................................................................................5
2. Background and Prior Work ............................................................................................6
   2.1. Hash Message Authentication Code (HMAC) .................................................................6
   2.2. Integer Encryption with HMAC .......................................................................................8
3. Design ................................................................................................................................11
   3.1. Design of the HTEE Scheme ..........................................................................................11
   3.2. Plaintext Bucket Decomposition ....................................................................................11
   3.3. Key Transformation .......................................................................................................12
   3.4. Element Key Transformation .........................................................................................13
   3.5. Bucket Key Transformation ..........................................................................................15
   3.6. Encryption .....................................................................................................................16
   3.6. Decryption .....................................................................................................................17
4. Security Analysis .................................................................................................................19
   4.1. HMAC Security ..............................................................................................................19
   4.2. HTEE Security ...............................................................................................................20
   4.3. HTEE Tamper Detection ..............................................................................................22
5. Implementation ...................................................................................................................23
   5.1. Overview .........................................................................................................................23
   5.2. Implementation Details ................................................................................................23
   5.3. Challenges Encountered ..............................................................................................24
List of Tables and Figures

<table>
<thead>
<tr>
<th>Table/Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Overview of the HTEE scheme</td>
<td>5</td>
</tr>
<tr>
<td>Figure 2</td>
<td>HMAC operation</td>
<td>7</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Overview of original HMAC process</td>
<td>8</td>
</tr>
<tr>
<td>Figure 4</td>
<td>HMAC and HASH Function input/output details</td>
<td>14</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Element key transformation function</td>
<td>14</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Bucket key transformation function</td>
<td>16</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Detailed encryption operation</td>
<td>17</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Detailed decryption operation</td>
<td>18</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Performance comparison of AES vs. HTEE methods</td>
<td>29</td>
</tr>
<tr>
<td>Figure 10</td>
<td>HTEE performance difference across bucket sizes</td>
<td>29</td>
</tr>
<tr>
<td>Table 1</td>
<td>Summary of features for tamper detection</td>
<td>3</td>
</tr>
<tr>
<td>Table 2</td>
<td>Average performance across bucket sizes</td>
<td>27</td>
</tr>
<tr>
<td>Table 3</td>
<td>Detailed performance for bucket sizes</td>
<td>28</td>
</tr>
<tr>
<td>Table 4</td>
<td>Complexity of HTEE and Original schemes</td>
<td>31</td>
</tr>
<tr>
<td>Table 5</td>
<td>Performance comparison among HMAC encryption methods</td>
<td>31</td>
</tr>
</tbody>
</table>
1. Introduction

1.1. Motivation

Increasingly databases are used to store a wide variety of sensitive data ranging from personally identifiable information to financial records and other critical applications. The volume and importance of sensitive data stored and processed by computers is constantly growing, and this data must be protected from unauthorized disclosure or modification. Confidentiality and integrity of this sensitive data must be maintained for legal, regulatory or fiscal reasons [22, 23, 26]. Confidentiality is a security goal that ensures that sensitive data is not revealed to unauthorized individuals, and integrity ensures that sensitive data is not corrupted or updated by unauthorized individuals. Integrity can also be referred to as tamper detection, a term used throughout this paper. There are a wide variety of problem domains where sensitive data must be secured, including web systems, archive systems, and systems that process sensitive information. Due to the increasingly large volume of sensitive data and the wide range of problem domains, a variety of solutions for confidentiality and integrity are of interest to suit particular situations [21, 22, 26].

The goal of this project is to provide confidentiality and tamper detection in a database environment. Existing work supports tamper detection and integrity for database systems using techniques such as access control, auditing, file system controls and other methods [21, 22]. Intrusion detection programs such as Tripwire and Samhain support tamper detection for the overall system. Additional related work includes the forensic analysis of database tampering [23], where a trusted notarization process is used to detect tampering and forensic analysis is applied after updates are detected. Some techniques apply encryption and authentication in parallel to provide confidentiality and integrity [24, 25]. Unlike these techniques, this project uses an encryption scheme based on the keyed Hash Message Authentication Code (HMAC) [3, 12] for confidentiality and integrity. Existing work makes use of HMAC for integrity but it is not typically used for encryption and confidentiality. One exception is presented in [1], which investigates HMAC as an encryption function.

The encryption scheme used for this project offers tamper detection and confidentiality directly in the encrypted data field rather than externally or at the system level. Cryptography provides several standard algorithms that support confidentiality and integrity in the encrypted data field, including symmetric and asymmetric encryption algorithms for confidentiality and hash digest or signature algorithms for integrity. Combining these solutions can require detailed processing by the end user, and may not be ideal for all problem domains. Objectives for this project’s encryption scheme include making it simpler
to implement both confidentiality and integrity in a database, improving the efficiency of the encryption operation over standard solutions, and researching the application of keyed-HMAC for encryption and key generation.

1.2. Problem Summary

The standard solutions for data confidentiality and integrity using cryptographic functions can be improved for some problem domains. The concept of data integrity or tamper detection relating to this project is specific to a database environment. In a database record sensitive data is usually paired with information that uniquely identifies the record such as primary key or hash digest. Each row in a database table contains a combination of uniquely identifying information and sensitive data, and this relationship must be preserved from encryption through decryption. If the relationship between uniquely identifying information and sensitive data changes then the data has been tampered with, this can happen while it is encrypted. In these situations the integrity of the data is lost. For example, if an attacker transposes encrypted values for account balance, the change must be detected.

Typically encryption algorithms such as the Advanced Encryption Standard (AES) or RSA provide strong confidentiality but don’t provide integrity and hash digest algorithms such as Secure Hash Algorithm (SHA) or Whirlpool provide integrity without confidentiality [16]. Traditional methods to obtain both confidentiality and integrity involve combining encryption and digest algorithms. A challenge with hash digest functions is that an attacker can freely recalculate and update the digest after changing the data. Once a digest is computed it must be stored in a trusted location or encrypted so it cannot be updated. Message authentication codes such as HMAC [3, 12, 13] provide an alternative to traditional hash digests. Message authentication codes provide the function of a digest that is protected from unauthorized update with a secret key, but they normally are not used for encryption.

Symmetric key encryption and hash digests can be combined in order to provide a standard method for confidentiality and tamper detection in a database system. An example of this solution is to compute a hash digest of a data record with an algorithm such as SHA-1, and secure both the digest and the sensitive data using AES encryption. When decrypting the sensitive data fields, the hash digest is recomputed and compared against the original digest. If the digests differ, then some change has occurred to the sensitive data in relation to the rest of the data record. In this example the use of AES encryption provides strong confidentiality and the secured hash digest provides integrity and tamper detection.

The standard solutions to the tamper detection problem, based on AES encryption, rest primarily on the end user or database administrator. The user defines the solution based on the schema and records of the database. Standard functions for AES encryption and hash
digests can be used, but the end user must build a custom process to compare digests and determine the validity of records. The difficulty of combining confidentiality and integrity in this situation could discourage the use of these techniques in some applications. In addition to complexity, the efficiency of encryption for standard solutions can be improved by using an HMAC based process. For large database systems and data archival, efficient encryption is an important feature.

1.3. Project Overview

This project presents a HMAC based encryption scheme that can provide confidentiality and tamper detection for positive integer data. This scheme is an improvement to the HMAC based integer encryption concept presented in [1]. Specific improvements include efficiency and tamper detection. The scheme is implemented in the PostgreSQL database environment [20], and the developed process is named “HMAC based Tamper Evident Encryption”, referred to as HTEE in this paper.

The HTEE process is simpler to use than the standard AES with SHA-1 solution, and more efficient for encryption. However this process is slower on decryption than AES with SHA-1, and the security of this scheme is dependent on the security of the underlying hash function. In general it is understood that security for this encryption scheme is not as strong as the AES algorithm, however it does provide significant confidentiality as discussed in Section 4 of this paper. Table 1 shows a comparison of features for the HTEE solution and two standard AES based solutions to the database tamper detection problem.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Encryption Strength</th>
<th>Tamper Detection</th>
<th>Simple Usage</th>
<th>Encryption Efficiency</th>
<th>Decryption Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>High</td>
<td>No</td>
<td>Yes</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>AES &amp; SHA-1</td>
<td>High</td>
<td>Yes</td>
<td>No</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>HTEE</td>
<td>Medium/High*</td>
<td>Yes</td>
<td>Yes</td>
<td>Fast</td>
<td>Slow</td>
</tr>
</tbody>
</table>

* Security of the HTEE scheme is variable and relies on the hash algorithm used. See Section 4 for more information.

Applications where the HTEE process is preferred against AES with SHA-1 include situations where simplicity and encryption efficiency are desired, and where AES is not required for regulatory data protection standards. For example, in a system that archives a large number of financial transaction records, encryption efficiency is important and tamper detection is critical. If an archived transaction states that an account received a deposit of $5.00, this value must be static so that an attacker cannot change it to $5,000.00. In this situation, the HTEE features of efficient encryption, strong tamper detection, and simple
operation are preferable to AES with SHA-1. This is particularly true if decryption is rarely needed, and if it is acceptable for tamper detection to be provided at the time of decryption. When used to store dollar amounts, the HTEE implementation must be limited to integer values, or dollar amounts must be multiplied by 100.

The HTEE scheme is a symmetric encryption process that relies on a secret key and processes positive integer values. The integer plaintext values are decomposed into components, or buckets, using modulus arithmetic. The buckets have a fixed size of 1,000, so integer values are decomposed into the value of the ones, thousands, millions, billions, etc. places. The plaintext buckets are encrypted using the HMAC function, where the hash digest represents the ciphertext. The secret key is modified for each plaintext value and each bucket value using a specific transformation process resulting in a different key for every HMAC operation. The key transformation process is based on a unique value related to the sensitive data, such as a database primary key. A primary goal of the HTEE process is the detection of unauthorized updates or tampering with ciphertext data, especially when valid ciphertext values are interchanged. The key transformation process ensures that this goal is met and ciphertext values can’t be changed without detection.

The decryption process is similar to the encryption process and includes the same key transformation sequence. Because the HMAC function produces a one-way hash digest, it is not trivial to reverse the operation. In order to find the correct plaintext for each bucket’s digest value an exhaustive search is performed across all 1,000 possible bucket values, calculating the HMAC digest of each one until a match is found. The search is repeated for all buckets in a plaintext value, and the modulus decomposition is reversed to obtain the original plaintext value. Any unauthorized updates to ciphertext data are detected in the decryption step by a failure to find a matching HMAC digest. Figure 1 shows a summary of the encryption process, including bucket decomposition, key transformation, and the HMAC digest function. Decryption is similar, but rather than a single bucket value HMAC operation there are up to 1,000 operations plus a comparison function.
1.4. Encryption Example

Consider the following example to illustrate the concept of the HTEE encryption scheme. A database record contains two fields, a primary key \{ID\} and a sensitive data value \{DATA\}. The primary key value is not encrypted because it is the index and is not sensitive data, but the sensitive data field is encrypted and needs to be protected from tampering. Two rows are included for simplicity:

- **Row1**: ID = 1000; DATA = 123456
- **Row2**: ID = 1001; DATA = 654321

The two data fields are decomposed into buckets of size 1,000 numbered from most significant to least significant. The resulting bucket values for each row are:

- **Row1**: bucket1 = 123; bucket2 = 456
- **Row2**: bucket1 = 654; bucket2 = 321

A 512 bit original secret key is used for encryption; this key is encoded in base64 format as:

- `fwWe6MNL5WC9gCfVbUsuFLeX8IfwKbnkWnlKhj5Txs20ds+VkmKS73AeFt0EsXy+zmfWEsy0EaKSx/oYMSmRA==`

The key transformation process modifies the original secret key four times, once for each row and bucket value. The resulting transformed keys, encoded in base64 format are:
The bucket1 and bucket2 values from each row are processed through HMAC with their respective keys to generate two digests, which are combined to form the final ciphertext.

The digest values and ciphertext, encoded in base64 are:

- **Row1, Bucket1:** MK5HUyCX1PyFGoVbKhlj16c8/1A=
- **Row1, Bucket2:** glAcZzmbDL8xRGwg23QBa5/mYuA=
- **Row1 ciphertext:** MK5HUyCX1PyFGoVbKhlj16c8/1A=glAcZzmbDL8xRGwg23QBa5/mYuA=
- **Row2, Bucket1:** ZiuYtd9t8Vnlh5ldqZjv57sTe2k=
- **Row2, Bucket2:** uk/ACtScX2oxJUpyEpDPWSPCXXqk=
- **Row2 ciphertext:** ZiuYtd9t8Vnlh5ldqZjv57sTe2k=uk/ACtScX2oxJUpyEpDPWSPCXXqk=

For this example, the following pairs represent the final plaintext and ciphertext data:

- **Row1:** ID = 1000; DATA = 123456; CIPHER = MK5HUyCX1PyFGoVbKhlj16c8/1A=glAcZzmbDL8xRGwg23QBa5/mYuA=
- **Row2:** ID = 1001; DATA = 654321; CIPHER = ZiuYtd9t8Vnlh5ldqZjv57sTe2k=uk/ACtScX2oxJUpyEpDPWSPCXXqk=

On decryption, the same key transformation process is used to obtain the four listed bucket keys. At each step, all 1,000 bucket plaintext values (0-999) are processed through HMAC to find a match to the digest value for the bucket. If a match is found, the decryption is successful. If a match is not found then the data has been tampered with. For instance, if the row1 ciphertext is transposed with the row2 ciphertext, or if the primary keys are switched, then the decryption process will not find a match and the tamper is detected.

2. **Background and Prior Work**

2.1. **Hash Message Authentication Code (HMAC)**

HMAC [3, 12, 13] is a process that uses a secret key and a hash algorithm such as MD5 or SHA-1 to generate a message authentication code, also referred to as a digest. This authentication code securely provides data integrity and authenticity because the secret key is
required to reproduce the code. Digests for normal hash functions can be reproduced with no such constraint. This process is symmetric, so two parties communicating with HMAC must share the same secret key. By using a hash algorithm in conjunction with a key, the HMAC function prevents an unauthorized user from modifying the message or the digest without being detected. This can protect against man-in-the-middle attacks on the message, but it is not designed to encrypt the message itself; only protect it from unauthorized update. The HMAC function was published in [3], which includes analysis and a proof of the function’s security, and it is standardized in FIPS PUB 198 [12]. HMAC is defined as a function that takes a key and a plaintext message as input and produces a binary authentication code, or digest, output. Any hash algorithm can be used including MD5, SHA-1, SHA-256, Whirlpool, etc.

The HMAC algorithm defines two padding constants, the inner pad and the outer pad. The inner and outer pads have values (0x3636...) and (0x5c5c...) respectively, each expanded to the block size of the hash algorithm. The secret key is a set of random bytes equal to the length of the block size. Smaller keys can be used, but will decrease security [3]. To calculate the HMAC digest, first the exclusive-or of the key and the input pad is found. This result is appended to the beginning of the message to be processed. The result is then hashed with the chosen hash algorithm, producing an intermediate digest. In the next step, the exclusive-or of the key and the output pad is found, and that result is appended to the beginning of the intermediate digest. The result is hashed again, producing the final message authentication code. The combination of the inner pad and outer pad with the secret key effectively generates two different keys, which adds additional security. This operation is summarized in Figure 2, where {⊕} denotes exclusive-or, {++} denotes concatenation, {K} is the secret key, {m} is the message, and {H} is the cryptographic hash function [13].

\[
\text{HMAC}(K, m) = H((K \oplus \text{ipad}) ++ H((K \oplus \text{ipad}) ++ m)).
\]

Figure 2 - HMAC operation

Each calculation of the HMAC authentication code requires running the underlying hash function twice. The output of HMAC is a binary authentication code, equal in length to the hash function digest. This code can only be reproduced with the same key and message, allowing an authorized individual to authenticate the message. The security of HMAC is directly related to the underlying hash function used, so it is weakest with MD5, moderate with SHA-1, and strong with SHA-512 or Whirlpool. Forgery and key recovery attacks threaten HMAC, but these generally require a very large number of message/digest pairs for analysis. A beneficial feature to the HMAC function is that it can be combined with a stronger underlying hash algorithm if security is a concern, which will defend against the attacks presented to date [3, 12]. A detailed discussion of HMAC security and its impact on
the HTEE scheme is presented in Section 4 of this paper. All HMAC functions used for this project are implemented using the SHA-1 hash algorithm; however the security of the HTEE scheme can be improved by using other hash functions such as SHA-512 or Whirlpool. The use of HMAC-SHA1 specifies some data sizes that are important in the HTEE implementation. These include the 512 bit (64 byte) block size of SHA-1, which becomes the key size of the HMAC-SHA1 function and the 160 bit (20 byte) digest output of SHA-1 which becomes the authentication code output of the HMAC-SHA1 function.

2.2. Integer Encryption with HMAC

The HTEE encryption scheme developed, researched and implemented for this project is based on a scheme presented in [1], and provides several improvements over that method. A detailed analysis and discussion of this original scheme is available in [2]. The original scheme [1] uses integer decomposition, HMAC for encryption, and decryption via exhaustive search, all concepts that the HTEE scheme is based on. A summary of the original HMAC encryption scheme’s process is shown graphically in Figure 3. There are several differences between the original scheme and HTEE. For example, the original scheme uses only two buckets for all plaintext values, and encryption is achieved with recursive HMAC iterated up to the bucket value. Also the original scheme does not combine related data with the plaintext data, so it cannot be used for tamper detection.

![Figure 3 - Overview of original HMAC process](image-url)
The original encryption scheme takes a positive integer input as plaintext, and first computes the remainder \( r \) with the formula \( r = m \mod S_b \), where \( m \) is the plaintext and \( S_b \) is a predefined bucket size. After calculating the remainder, the bucket ID \( I_b \) is found using the formula \( I_b = (m - r) / S_b \). As an example, when processing the integer 485,321 with a 20,000 bucket size, the remainder is 5,321 and the bucket ID is 24. The bucket ID and the remainder are encrypted separately in the next phase. The selection of bucket size is an important factor this encryption scheme, as efficiency and validity are affected if the bucket size is incorrect for the problem domain. The bucket size also controls the largest plaintext integer value that can be processed. In addition to the bucket size, the maximum bucket ID \( M_b \) is defined, and in most cases the values are equivalent, \( S_b = M_b \). The maximum plaintext value that can be processed with this scheme is equal to \( S_b * M_b \). So in the case of a bucket size of 20,000 and maximum bucket ID of 20,000, the scheme can process values up to \( 4 \times 10^8 \). The limitation of maximum bucket ID is required for the correct operation of the decryption function. The maximum bucket ID and bucket size can be determined from domain knowledge or the data type being encrypted. Typically a value of the square root of the maximum plaintext value is used for maximum bucket ID and bucket size.

After decomposition into the remainder and bucket ID, the values are encrypted. Inputs into encryption include a secret key, a seed value, the plaintext bucket ID and the remainder. Keyed HMAC is used recursively to encrypt the bucket ID and remainder independently. The encrypted bucket ID is found by calculating the HMAC function repeatedly \( N \) times, where \( N \) is equal to the bucket ID. On the first iteration, the secret key and a predefined seed value are used as input into the HMAC operation. For successive iterations, the output of the previous HMAC is used as input into the next iteration along with the secret key. This is repeated until bucket ID iterations are completed. For example, in the case of bucket ID equal to 24, HMAC will be executed recursively 24 times, using a predefined seed value for the initial message. The result is labeled \( \{T(I_b)^{K}\} \), designating the transformation on bucket ID \( I_b \) using key \( K \). In this way, the bucket ID is not directly encrypted, but the execution of recursive HMAC is based on the value of the bucket ID.

The encrypted value for the remainder is found in a similar operation, differing only in the secret key that is used. Each plaintext decomposition operation forms a related bucket ID and remainder pair. When encrypting the remainder value, the corresponding bucket ID is appended to the beginning of the secret key to form a new key. After finding the new key, the recursive HMAC operation is the same. Beginning with the seed, the digest is calculated \( N \) times where \( N \) is equal to the value of the remainder. This result is labeled \( \{T(r)^{I_b \| K}\} \), designating the transformation on remainder \( r \) using the composite key \( I_b \| K \). As an example of this scheme, consider the encryption of integer 336,789 with a bucket size of
1,000. The bucket ID is 336 and the remainder is 789. If using the SHA-1 hash algorithm, a key of “999”, and a seed value of “test”, HMAC will be executed recursively 336 times for the bucket ID, and 789 times for the remainder. Both recursions use “test” as the initial HMAC message, but the bucket ID uses key \{K\} = 999 and the residual uses key \{I_b \mid |K\} = 336999. The resulting encrypted values, encoded in base64 format are \{T(I_b)^k\} = “2CI0b3pN8B8KbiCIUbKKo2ciRAc=” and \{T(r)^{ib\mid |k\} = “PynDpvSFSSUZCqk3yY8J2g3Ks4=”. Note that the output in this situation is two 28 character base64 encoded strings, which is a result of the 160 bit digest output of the SHA-1 hash used with HMAC in this example.

To decrypt, or reproduce the plaintext from the ciphertext an inverse transformation is defined. Because the algorithm uses a hash as the basis of its encryption a direct inverse cannot be calculated. The inverse transformation must search through potential bucket ID and remainder values. The inverse transformation uses the set of possible bucket IDs as a range for the search process, hence the requirement to define the maximum bucket ID \{M_b\}. The first step for the decryption transformation is finding the bucket ID of the ciphertext data. This operation will reproduce the value of \{I_b\} from ciphertext \{T(I_b)^k\}. The same seed and key value from encryption are used in the HMAC operation, and this operation is executed N times, where N is the number of possible buckets in the domain. For example, if using a bucket size of 2,000 in a domain where the maximum data value is 1,000,000, there are 500 possible bucket values and HMAC is executed 500 times. In this way the upper limit of allowable data values must be known in order to provide a limit to the HMAC search loop. While the N iterations of HMAC are calculated, the input for each calculation is based on the output of the previous iteration. Each time, the resulting value is compared against all encrypted bucket IDs for a match. If a match is found, the bucket ID plaintext is equal to the number of loops executed in the search.

Once a bucket ID is found, a similar search is made for the remainder value. Again, a new key is constructed by appending the decrypted bucket ID to the beginning of the secret key, and HMAC is calculated N times, where N is equal to the bucket size. The bucket size defines all possible remainder values. Once a match is found between the HMAC output and the encrypted remainder value, the plaintext remainder is equal to the number of loops executed in the search. After finding the plaintext bucket ID and remainder values, the modulus decomposition is reversed to generate the original plaintext from the decrypted bucket ID and remainder. The plaintext value \{m\} is found using \{m = I_b \ast S_b + r\}, where \{I_b\} is the decrypted bucket ID, \{r\} is the decrypted remainder, and \{S_b\} is the bucket size.

Notable points from the original scheme include the basic idea of integer value decomposition, key transformation, exhaustive decryption search, and HMAC as encryption function. Issues identified with the original scheme include the problem that two buckets (remainder and bucket ID) decrease efficiency for large integer values, the key
transformation only occurs on the remainder value, rather than the bucket ID, and the highly recursive use of HMAC is inefficient [2]. The HTEE process developed for this project improves each of these points by defining a general rule for integer decomposition that improves performance, defining a secure key transformation process, and using HMAC as encryption while removing the recursion requirement. In addition, tamper detection is added by making the ciphertext dependent on other related data, a feature not present in the original scheme.

3. Design

3.1. Design of the HTEE Scheme

The HTEE scheme developed and implemented for this project makes several improvements to the original HMAC integer encryption process presented in [1]. The HTEE process is similar to the original scheme in that positive integer values are processed, these values are decomposed into components, also called buckets, and the bucket values are processed through HMAC for encryption. The combination of HMAC output for all bucket values creates the ciphertext. The decryption step calculates the HMAC digest for all possible bucket values, where a match between calculated digest and ciphertext data indicates the correct plaintext result. HTEE includes a key transformation process that ensures each bucket of each plaintext uses a different encryption key. Pseudocode for the primary HTEE procedures is provided in Appendix A.

3.2. Plaintext Bucket Decomposition

The first step of the encryption process is decomposition of the integer plaintext input. The original scheme used a single modulus operation, with the quotient and the remainder representing two buckets for HMAC processing. This simple decomposition causes efficiency problems on decryption with large integer values, such as values above one million. For the HTEE scheme, the integer plaintext value is decomposed into multiple buckets of size 1,000. The number of buckets used for a given plaintext is calculated with \( \text{buckets} = \lfloor \log_{1000}(\text{Plaintext}) \rfloor + 1 \). For example, a plaintext integer value of 14 trillion (14x10^{12}) will use five buckets. Because each bucket produces one HMAC digest value, larger plaintext values will produce a larger ciphertext. In order to avoid leaking information about the plaintext’s order of magnitude, a domain specific maximum number of buckets are defined. Small plaintext values are processed with the larger number of buckets, but the extra buckets use a value of zero for encryption. By using more buckets of smaller sizes, the
decryption operation becomes much more efficient because a smaller number of HMAC searches must be performed. The tradeoff to this configuration is that additional digest ciphertext data is produced.

As an example of the efficiency difference, consider plaintext value of 2,412,345,678. If using two equal sized buckets as in the original HMAC encryption scheme, each should be 50,000 in size, but the HTEE process will use four buckets of size 1,000. Specifically, the two bucket solution decomposes the plaintext into bucket values of (48246; 45678) while the four bucket solution uses bucket values (2; 412; 345; 678). When decrypting this value, the original scheme could potentially process HMAC 2*50,000 or 100,000 times to find the plaintext match. The HTEE scheme could process HMAC 4*1,000 or 4,000 times to find the plaintext match. This represents a 25-fold decrease in the processing load required, while it only doubles the amount of ciphertext data stored. A more detailed analysis of performance differences is presented in Section 6 of this paper. An important point regarding the HTEE scheme is the size of the problem domain. If a system processes numeric data up to sixteen digits, it would be require six buckets, but a system that processes numbers up to nine digits would only require three buckets. By planning the use of the HTEE process around the maximum integer length for the problem domain, additional improvements to performance can be achieved when fewer buckets are required. This assumes that the maximum length of data in the problem domain is not information that must be hidden from an unauthorized party. This could be the case for account numbers, transaction amounts or other information with standard formatting.

3.3. Key Transformation

The second step of the encryption process is key transformation, which prepares distinct secret keys for the encryption of each bucket value. The original scheme only modified the secret key for the remainder or second bucket value, while the bucket ID always used the original secret key. This solution has several problems that motivated changes for the HTEE scheme. The first and primary concern is that equal values for plaintext bucket IDs will result in equal ciphertext digests, potentially providing information to an unauthorized individual. The second concern is that ciphertext data can be interchanged without being detected because the process does not rely on information beyond the original plaintext and secret key value.

The HTEE scheme improves the original process and adds tamper detection by defining two key transformation functions, an element transformation and a bucket transformation. The element key transformation creates a secret key for each plaintext value processed, ensuring that equal plaintext values do not have equal ciphertext. The element
transformation is seeded with information relating the plaintext data to its environment, which provides tamper detection. The bucket key transformation produces a secret key used on each decomposed bucket value of a given plaintext so equal bucket values do not result in equal ciphertext. The bucket key is the effective encryption key because only decomposed bucket values are encrypted. The method of key transformation used for bucket values also contributes to tamper detection because it is a continuation of the element key process. Both the bucket and element transformations use the HMAC function to generate new secret key data. For its use here as a key transformation function, HMAC is considered a pseudo-random value generator. Research supports HMAC as a pseudo-random function, as discussed in [3, 4, 9, 11]. The key transformation functions used for HTEE provide a critical security feature that makes analysis of the ciphertext output more difficult for an attacker. By using different secret key values for each encrypted value, there is an additional layer of analysis required in order to reproduce the original secret key.

3.4. Element Key Transformation

The initial design of the HTEE scheme defines two types of element key transformations; a unique value based transformation and an order based transformation. The unique value based transformation is the preferred method, particularly for database processing. The unique value based key transformation constructs an element key using the original secret key and uniquely identifiable data related to the plaintext value. Usually the unique value is the primary key of the database record, but any unique data can be used including a hash digest. The basic requirement is that the value will not be repeated for another plaintext. The first step of the transformation calculates the hash digest of the unique value with the SHA-1 algorithm, and then uses the digest as input into the HMAC function alongside the original secret key. The output of this HMAC operation is used for the first 20 bytes of the element key, and it is used as input into another HMAC operation with the original secret key. The output of the second operation is used for the second 20 bytes of the element key, and it is processed through HMAC again. This process repeats until four recursive HMAC operations are executed, outputting 80 bytes of key data. The output is then truncated to 64 bytes, producing the element key. This process is depicted graphically in Figure 5, using HMAC and hash functions outlined in Figure 4.

The unique value method for element key transformation will generate a distinct and secure key for each plaintext value. An attacker cannot reproduce the key if given the unique value, because the process is secured with the HMAC function and secret key. The unique value based key transformation process is important for HTEE tamper detection because it incorporates information related to the plaintext value with the encryption of the value. The
result is that decryption of the ciphertext is dependent on the unique value, and any changes between ciphertext and unique value can be detected.

**Figure 4 - HMAC and HASH Function input/output details**

The second method for element key transformation is order based, and is not dependent on information related to the plaintext value. This method ties the ciphertext data to the order of processing, and will detect ordering changes of stored ciphertext during
decryption. The mechanism is iterative and replaces 20 bytes of the element key for each plaintext value. The HMAC digest of the previous element key is computed using the original secret key and the 20 byte output is appended to the beginning of the element key, and the key is truncated to 64 bytes. The first iteration uses the original secret key in place of the previous element key. The order based transformation is less preferred than the unique value transformation because information related to the plaintext is not used in the process, only the position of the plaintext and ciphertext must match. Although the utility of this method is minimal for a database environment it is retained for potential use in the command line, flat file HTEE tool and for problem domains where such an approach could be appropriate.

3.5. Bucket Key Transformation

The second key transformation function used by HTEE is the bucket key transformation. The HTEE process uses a different key for each bucket’s HMAC function so that buckets with equal values do not have equal digests, and to support the tamper detection process. The bucket key transformation is iterative, and 20 bytes of the bucket key are replaced for each bucket processed in a plaintext. The first bucket key is equal to the element key generated for the plaintext value. Each succeeding bucket key is generated by processing the bucket’s HMAC encryption ciphertext through HMAC again with the original secret key. The result of this HMAC operation is appended to the beginning of the bucket key, and the result is truncated to 64 bytes resulting in the succeeding bucket key.

The bucket key transformation is summarized graphically in Figure 6. The function presented in Figure 6 depicts both the calculation of the bucket ciphertext as well as the transformation of the bucket key. The initial bucket key shown in Figure 6 is equal to the element key for the first iteration, and the prior bucket key for other iterations. The bucket key transformation makes encryption keys dependent on both the unique value used to generate the element key, and the order of processing for the bucket values. The combination of element and bucket key transformations produces distinct keys for each plaintext bucket value provided that differing unique values are input. The only cases when the key generation process will not result in distinct keys are for hash collisions based on the unique value data, which are extremely rare cases.
Figure 6 - Bucket key transformation function

### 3.6. Encryption

The design concepts discussed thus far include bucket decomposition, element key transformation and bucket key transformation representing the primary features of the scheme. The final encryption step of the process calculates the HMAC digest using the key and plaintext values for each bucket. The digests are concatenated to form the ciphertext output. The calculation of ciphertext for each bucket value is shown in Figure 6, because it relates to the bucket key transformation function. A detailed summary of the entire HTEE encryption process for a single plaintext value is shown in Figure 7, including decomposition, key transformation and HMAC encryption steps.

The HTEE encryption operation is a very efficient process regarding computation time, because the HMAC function is executed a small number of times. For example, when processing a plaintext value using four buckets, HMAC will be invoked twelve times. However, the decryption process for HTEE presents a performance challenge due to the need for exhaustive searching across possible plaintext values. In the example of a four bucket plaintext, HMAC could be executed up to 4,008 times. More information on this performance result is presented in Section 6 of this paper.
3.6. Decryption

The HTEE decryption operation is similar to the encryption operation, particularly with the key transformation functions. The same progression of element keys and bucket keys is calculated, except these keys are used for a search across all plaintext bucket values. The first step in the decryption process is splitting the concatenated ciphertext string into individual bucket digests. Then the key transformation process is used with the unique value data (which cannot be encrypted) to find the same bucket key values used during encryption.
The process then iterates through all possible bucket plaintext values, 0 through 999, calculating the HMAC digest for each one with the bucket key. The intermediate digest is compared with the stored bucket digest, if the values match then the current iteration is the bucket’s plaintext value. If no records from 0 through 999 match the bucket digest, then some corruption or tampering of the ciphertext has occurred. This step is the critical tamper detection operation for HMAC; the absence of a correct decryption match indicates that the ciphertext data or unique value has changed since encryption. Once all bucket plaintext values are identified, the modulus decomposition is reversed using a calculation such as \( \text{value} = \text{bucket1} \times 1000^2 + \text{bucket2} \times 1000^1 + \text{bucket3} \times 1000^0 \), in the case of a three bucket (9x10^8) plaintext. A detailed diagram of the decryption operation is depicted graphically in Figure 8. The descriptions for functions that are also used in the encryption operation are omitted.

**Figure 8 - Detailed decryption operation**

\[
\text{HMAC}(B, K), \ f_3(K, U), \ f_4(K, C)
\]

These functions are documented with the encryption operation

**Compare (C’, C)**:

Compare \([C’_{30}]\) and \([C_{30}]\). If match, then \([B’_{30}]\) is the plaintext component, equal to \([B_{30}]\).
4. Security Analysis

4.1. HMAC Security

The security of the HTEE scheme is primarily based on the security of the HMAC function, because HMAC is used for both key transformation and encryption. Existing work has established that the security or cryptographic strength of the HMAC function is directly related to the security of the underlying hash function on which it is based [3, 4, 5]. Although recent findings on collision attacks have invalidated the use of the MD5 hash algorithm and decreased confidence in the SHA-1 algorithm, these attacks have limited impact on HMAC security [3, 4, 5]. Because the HMAC function uses an inner hash with a hidden key, it is more difficult to find collisions [5]. In addition, due to the outer and inner hash functions, an attacker cannot control input into the outer hash function which makes it difficult to attack the function [6].

The designers of the HMAC function proved its security given two features of the underlying hash function: that the hash function is weakly collision resistant, and that the hash compression operation is a pseudo-random function [3]. In response to attacks against the collision resistance of MD5 and SHA-1 the designers presented a further proof of HMAC security provided that the hash compression operation is a pseudo-random function, dropping the collision resistance requirement [4, 7]. The secret key reduces the effect that collision based attacks have on the HMAC function [4, 5, 11]. This improvement is significant to the security of HMAC, because it means that hash algorithms such as SHA-1 can continue to be used in HMAC processing [4, 5]. Beyond this assurance, one of the beneficial features of using HMAC is its extensibility to other hash functions. If additional security is required, the hash algorithm can be upgraded to a SHA-512, Whirlpool, or other strong functions [3]. For HMAC processing and the HTEE scheme, the only factors that will be impacted by changing the hash function are the key size, the output digest size, and processing cost.

While the strength of HMAC security is based on the compression operation of the underlying hash function, the measure of security is the difficulty to produce a forgery of the authentication code. If legitimate parties communicate with message / authentication code pairs \(\{M_1, A_1\} \) through \(\{M_n, A_n\}\), a forgery is the ability for an attacker to produce a new pair for \(\{M_x, A_x\}\) for a message not communicated legitimately [3, 6]. The attacker is able to see the legitimate message pairs, but not the secret key. There are several methods researched to produce forgeries in the HMAC function, the primary being the birthday attack. Although collisions of the underlying hash function are not a concern for the HMAC, it is still the case the HMAC output is a digest of a message and secret key input, and it can
produce its own collisions. It is possible for an attacker to observe messages \( M_1 \) and \( M_2 \) where \( \{ M_1 \neq M_2 \} \) but \( \{ A_1 = A_2 \} \). This probability of this occurrence is controlled by the birthday paradox, where a HMAC collision becomes probable after \( \{ 2^{n/2} \} \) message pairs are observed, where \( \{ n \} \) is the number of bits in the output digest [3, 5, 11, 15]. So a HMAC-SHA1 function would be susceptible to a forgery based on the birthday paradox after \( 2^{80} \) message pairs are observed. The birthday paradox is also used to find collisions in hash functions but with HMAC the attacker relies on a legitimate user to generate all \( 2^{80} \) digests. With traditional hash functions, the attacker can generate digests at will. Also the effect of a birthday attack is a forgery, and does not yield the secret key. A forgery could compromise a single record’s tamper detection capability, but it won’t threaten the entire database.

Beyond the birthday attack, full key recovery attacks are another important threat to the HMAC function. These attacks still appear infeasible, although some methods have efficiency better than brute force [6, 7, 10]. These methods have an underlying requirement of a very large number of HMAC message/authentication code pairs for analysis, more than are required for the birthday attack [6, 7, 10]. Several conclusions can be made from this analysis of HMAC security research. The first conclusion is that the HMAC function is secure from collision attacks presented in MD5 and SHA-1, and the attacker cannot generate potential collisions offline but is dependent on a legitimate user. The second conclusion is that key recovery is a difficult attack and is only feasible after a very large number of messages are analyzed. Furthermore, the secret key value and underlying hash function used in the HMAC process are the primary contributions to its cryptographic strength.

4.2. HTEE Security

In the context of the HTEE scheme, the HMAC operation is secure considering typical birthday attacks and key recovery attacks. Even if the HTEE scheme were used in a large scale environment, it would be unlikely that a single database table would handle over \( 2^{40} \) (approx. 1 trillion) records. Even with \( 2^{40} \) records and six buckets of HMAC digest data for each record, this is not close enough to the number of messages required to perform key retrieval or birthday attacks if HMAC-SHA1 is used [3, 6, 7]. Implementation can be customized to provide additional security such as using different secret keys periodically. Ideally a key management implementation would alternate keys after a large number of records were processed.

An additional consideration for security of the HTEE scheme includes the input of unique value and plaintext value as messages for the HMAC function. The data ranges for unique value can vary widely according to the problem domain, and the plaintext value will always have a small range due to the HTEE bucket decomposition limiting values to integers
As discussed earlier, each plaintext bucket value uses a different secret key generated from the bucket key transformation process. This provides a layer of defense for small values because any analysis of the ciphertext data will be challenged with constantly varying keys. However, the key transformation process begins with the unique value input which is known to the attacker since it cannot be encrypted in the database. A likely method for an attacker to pursue is attacking the key transformation function using the unencrypted unique values. The natural variation of the unique value is masked by the hash and recursive HMAC functions in the element key transformation. Considering the use of HMAC as a pseudo-random function, the variation in key values through the transformation process should be secure and unpredictable by an attacker. This is expected even if the unique value size is small, due to the pseudo-random feature of the underlying hash compression function. At the end-user level, additional data could be provided for the unique related value, thus expanding it beyond the range of small input values. For example, instead of using a four byte integer primary key as the unique value the user can concatenate a text string field that contains many bytes of data.

The structure of the HTEE scheme provides additional protection by obscuring internal values in a similar way to the inner and outer hash operations of the HMAC function. The layering of the HTEE scheme protects the secret key, and makes it more difficult for the attacker to perform analysis over the ciphertext. Consider that the attacker knows two values: the ciphertext output from HTEE, and the unique value input. The HTEE function can be written in a short format as \( \text{HTEE}(P,K,U) = \text{HMAC}(P, f_K(K,U)) \), where \( \{P\} \) is the plaintext value, \( \{K\} \) is the original secret key, \( \{U\} \) is the unique value, and \( \{f_K\} \) is the key transformation function. The \( \{f_K\} \) function is a combination of several HMAC steps as described previously, and produces intermediate keys \( \{I\} \). The attacker knows one item \( \{\text{Digest}\} \) in the relationship: \( \{\text{Digest} = \text{HMAC}(P, I)\} \), and one item \( \{U\} \) in the relationship \( \{I = f_K(K, U)\} \). It is difficult for the attacker to generate the intermediate key used with plaintext value \( \{P\} \), based on the above analysis of HMAC key recovery attacks [6, 7, 10]. It is also difficult for the attacker to identify the secret key \( \{K\} \) using the input message \( \{U\} \), because the result of function \( \{f_K\} \) is not known. In this way, the plaintext is protected by varying key values, and the secret key is protected because the intermediate key values are hidden.

This analysis shows that the HTEE scheme is at least as secure as the base HMAC function, using an appropriately random key, and a secure implementation. In addition, security can be strengthened with a stronger underlying hash function, and regular key replacement. However, until a mathematical proof of this scheme is presented, critical or regulatory mandated cryptography uses should continue to use the AES standard.
Conceptually, the HTEE scheme will provide solid confidentiality for non-regulated uses of numeric encryption.

### 4.3. HTEE Tamper Detection

In addition to confidentiality, tamper detection is an important feature of the HTEE scheme. In the context of database records, tampering is defined as a failure in data integrity between the ciphertext data and the remainder of the data record. The data integrity relationship can be defined between the record’s primary key and the plaintext/ciphertext value, or additional fields can be combined into the integrity relationship. If every field of a database record or a hash digest was input into the HTEE function, then the ciphertext could detect changes in any data of the record.

An attacker can try to modify the data record in three possible ways: Case 1) Make a random change to ciphertext, Case 2) Interchange two ciphertext values and Case 3) Make a change to the unique value. The tamper detection feature of HTEE will detect each of these changes through the decryption viability test. If the modifications in Case 1 or Case 2 were employed, the unique value would be unchanged and the key transformation sequence for decryption would be identical to the encryption operation. Each step in the decryption search would iterate through the 1,000 possible plaintext values, but none of the HMAC digests would match the stored value. The probability of a false positive would be extremely small, approximately $3.42 \times 10^{-43}$, based on the birthday attack with 1,000 values [15, 16]. This result is obtained with the formula $P = 1 - e^{-(k^2/2N)}$, where $k$ is the sample size, equal to 1,000 and $N$ is the number of possible values, equal to $2^{160}$ for SHA-1. This is the probability that the same key value will have a conflicting authentication code within 1,000 plaintext values. If the modification in Case 3 was employed, the key transformation sequence would be changed resulting in a similarly improbable collision. The new key transformation and a value between (0-999) would have to collide with the original transformed key and a value between (0-999). In addition to these very improbable collisions, the multiple bucket solution makes the probability even smaller. If the ciphertext or unique key was changed, each bucket digest would have to produce a collision for the tamper to be undetected. The HTEE process will flag a value as tampered if any of the bucket values cannot be decrypted. Based on this analysis, HTEE is very strong with tamper detection, provided that the unique values input into the process are unique for each record.

From this analysis of HTEE confidentiality and integrity, the process is ideal for situations where tamper detection of stored data is essential and confidentiality is desired. In situations where high level encryption is required by law or regulation, AES is still the
recommended standard. However, HTEE can be used for numeric items that are moderately sensitive and susceptible to tampering, such as financial balances and transaction amounts.

5. Implementation

5.1. Overview

The HTEE scheme was implemented for this project to validate the designed algorithm, evaluate performance, and provide a tool that could be used for future applications. The implementation has two varieties, a command line program designed for flat file processing, and a database add-on for the PostgreSQL database management system. The command line program is used for debugging, program validation and testing while the PostgreSQL add-on is the primary method to use HTEE for encryption and tamper detection. The implementation was developed and compiled on Windows XP using the Microsoft Visual C++ 2008 Express Edition compiler. The PostgreSQL add-on was compiled against server versions 8.3.8 and 8.4.1.

Both programs are implemented in the C language, chosen because the PostgreSQL system and libraries are implemented in C and extensibility is supported for the language. The PostgreSQL add-on is installed into the database as a function which is conveniently invoked through queries, such as “SELECT htee_enc(data,unique) FROM test” for encryption. The add-on is configured through a file in the server’s data directory, which allows specification of a secret key and a maximum number of buckets. The secret key must be stored in base64 encoding in a file accessible to the server process. The maximum number of buckets is an important processing parameter that defines how many buckets the program can decompose integer values into, which controls the maximum range of plaintext values and efficiency. As the number of buckets processed increases, the program becomes less efficient but it can support a wider range of plaintext values. Valid values for the maximum number of buckets are one through six, due to the 8 byte integer limit near 9x10^18 preventing the use of a seventh bucket. Appendix B contains details related to the implementation including support for compilation and installation of the add-on.

5.2. Implementation Details

The implementation uses the HMAC operation with SHA-1 as underlying hash function and for the element key transformation. The use of HMAC-SHA1 specifies several parameter sizes that are important during implementation including the secret key size and the digest output size. The implementation uses base64 encoding for the input of secret key
and output of ciphertext data, making the input key 64 bytes or 88 base64 characters, and the output digest a multiple of 20 bytes or 28 base64 characters. The bucket size used for the implementation is 1,000, which provides the effect discussed previously of breaking numbers into buckets by order of magnitude such as millions, billions, trillions, etc. Plaintext values supported are positive integers in the range of 0 through 9.9x10^{17}. Values of 1x10^{18} and above are not supported because the range of the 8 byte integer prevents the use of a full seventh bucket. Plaintext values are represented internally as 8 byte integers to support mathematical calculations including modulus, power and logarithm.

The maximum plaintext values of 16 through 18 digits are processed with six buckets, 13 through 15 digits are processed with five buckets, and so forth with 1 through 3 digits processed with one bucket. Each bucket value is up to three plaintext digits (values 0-999) which are encrypted into 28 base64 encoded characters. A six bucket HTEE ciphertext would require 168 bytes of text data. This is a nine-fold increase in storage space when the plaintext is stored as a text string, or a twenty-one fold increase when the plaintext is stored as an 8 byte integer. However, the equivalent AES ciphertext requires 116 bytes of base64 text data in PostgreSQL, so HTEE is only a 44% increase over the AES requirement. The large increase in storage space is one of the costs of using the more efficient small bucket solution employed by HTEE. The other primary cost is decryption processing time. When processing plaintext values that are smaller than the maximum number of buckets, a value of zero is used for the larger bucket positions. This allows valid ciphertext to be generated, and disguises the magnitude of the plaintext from an attacker.

The PostgreSQL add-on installs two functions into the database, \{htee_enc\} for encryption, and \{htee_dec\} for decryption. The parameters for the encryption function are \{plaintext; unique value\}, and the parameters for the decryption function are \{ciphertext; unique value\}. In order to support the widest range of accurate numeric data, the PostgreSQL 64-bit Integer data type “bigint” is used for plaintext input and output, and the unlimited length “text” data type is used for ciphertext output and input. The unique value input is in the form of unlimited length “text” data type, and should be at a minimum the primary key of the database record, and at a maximum the concatenated data fields from the entire data record. A hash digest string can also be used for a unique input value. Since the database add-on is designed to handle plaintext data in the “bigint” format, and ciphertext data in the “text” format, these values should be stored in separate database fields, or input data should be cast to the correct data type.

5.3. Challenges Encountered
The implementation effort encountered several challenges, both in the command line and PostgreSQL program development. To focus effort on the HTEE design, refinement, analysis and related research, existing implementations of SHA-1 and HMAC were used [17, 18, 19]. This also helped to ensure the validity and efficiency of the hash implementation. Development of the command line program encountered problems with C language hash processing including null byte errors, memory management and data encoding for base64 and binary data. When using the C language for hash processing, care must be taken to use direct memory modification functions such as memcpy() rather than string functions like strcpy(), because null bytes in the hash output will truncate the string. Memory management is always an important topic in C programming as critical problems can occur if the boundaries of allocated buffers are not enforced. When processing binary data like that produced in hash functions, range checks for buffers should always be considered.

The implementation of the PostgreSQL add-on had additional challenges, including interfacing with the database server backend and compilation in a Windows environment. The PostgreSQL server is native to a Linux environment, and was only ported to Windows several years ago. With the complex variety of Windows libraries and compilation environments, it can be challenging to develop PostgreSQL extensions in this environment. In addition PostgreSQL defined data types and functions such as palloc() for memory allocation that can be challenging to use. For example, data structures in PostgreSQL have a built in size specification, VARHDRSZ which must be taken into account for memory allocation and referencing. After thorough research, debugging and testing efforts, several compiler configuration parameters and PostgreSQL C library modifications were identified that allowed creation of an operational add-on DLL [20]. Source code for the command line and PostgreSQL programs are available in [27], as is documentation to support compilation and use of the programs. Additional information about the implementation is presented in Appendix B, including changes made to PostgreSQL library files.

6. Testing

6.1. Test Structure

To validate the results of the HTEE implementation and to verify expected performance improvements, several tests were performed including comparisons to AES based techniques. Three encryption techniques were tested in the PostgreSQL system: 1) Raw AES encryption, 2) AES encryption with unique value data and 3) the HTEE encryption scheme. An additional test was run for the command line based HTEE program. Method 1, the raw AES encryption scheme, is straightforward and uses the delivered
PostgreSQL AES function with a secret key value. This method can detect random changes to ciphertext data, but it cannot detect other tampering such as the interchange of valid ciphertext values. For example, if two stored ciphertext values are switched, this technique will decrypt the values with no indication of the change. Method 2, using AES encryption with unique value data is a solution that adds tamper detection to the raw AES encryption. The approach used for AES tamper detection includes concatenating the unique value data with the plaintext data, separated by a semicolon, and encrypting the combined string. This technique could be implemented equivalently with a secure hash digest of the unique value. On decryption, the unique value is separated from the plaintext, and the plaintext is recovered. If the decrypted unique value differs from the current unique value, the data was tampered with and a warning can be made. The AES with tamper detection technique is more difficult to implement from the database user’s perspective. In order to decrypt the concatenated data and compare the original and decrypted unique values, additional processing is required. The HTEE encryption scheme used the primary key as unique value, and managed tamper detection internally. For both AES with tamper detection and HTEE schemes, a decrypted value of \{-1\} was used to indicate that a record was tampered with. Specific commands used during testing including SQL statements that highlight the complexity differences between the methods are available in Appendix C.

The testing process used six datasets, each composed of 20,000 randomly generated integers. The datasets were each configured with a different number of buckets, so one dataset had values between 0 and 999 (one bucket), another dataset had values between 1,000 and 999,000 (two buckets), etc. up to the six bucket, 18 digit numeric size. Performance was timed for the encryption, decryption and tamper detection operations. The tamper detection dataset was built by intentionally changing half of the ciphertext records, so that each even record \{n\} was set equal to the ciphertext of odd record \{n-1\}. The raw AES method did not detect the tampered records, but the AES with unique value and HTEE methods detected the tampered data.

### 6.2. Test Results

Performance results from testing are presented in tables below. Table 2 shows the average performance across all buckets, and Table 3 shows the detailed results with each bucket size displayed. The charts in Figures 9 and 10 depict the performance changes encountered as plaintext values increased. In the results shown below, “aes1” refers to method 1, raw AES encryption, and “aes2” refers method 2, AES with unique value concatenation. The HTEE method is divided into PostgreSQL version and command line version.
The testing results shown in Table 2 demonstrate the tradeoff in efficiency between the HTEE scheme and AES based schemes. The encryption operation for AES with tamper detection was about 4.5 times slower than the encryption operation for HTEE (15.8 seconds vs. 3.5 seconds). Conversely, the decryption operation for HTEE was about 4.1 times slower than the decryption operation for AES (75.4 seconds vs. 18.2 seconds). Processing for the tampered dataset is faster in HTEE because the program determines that a record has been tampered with if just one bucket cannot be decrypted, so it can move to the next record immediately. These performance numbers can be affected by implementation decisions.

One implementation decision that affects HTEE efficiency is related to how many of the 1,000 values for each bucket were searched against before a match is found. In order to improve decryption performance, once a match is found the remaining values between (0-999) are not searched. For example, decryption of a value 1,001,001 will be much faster than decryption of a value 9,999,999 because two of the buckets must iterate through 1,000 HMAC operations before a match is found. This implementation decision can open up a form of security hole, because the size of the plaintext buckets is related to the processing time taken. If needed, this can be disguised by always processing 1,000 searches instead of exiting. Another implementation decision to improve performance is to terminate the decryption search after a single bucket has been identified as tampered. In the case of an unauthorized update that swaps ciphertext values, the tampering can be detected when processing the first bucket. This can save significant time when processing datasets with multiple buckets.

### Table 2 - Average performance across bucket sizes

<table>
<thead>
<tr>
<th>Encrypt Method</th>
<th>Mode</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes1 postgres</td>
<td>encrypt</td>
<td>18.1</td>
</tr>
<tr>
<td>aes1 postgres</td>
<td>decrypt</td>
<td>15.3</td>
</tr>
<tr>
<td>aes1 postgres</td>
<td>tamper</td>
<td>18.3</td>
</tr>
<tr>
<td>aes2 postgres</td>
<td>encrypt</td>
<td>15.8</td>
</tr>
<tr>
<td>aes2 postgres</td>
<td>decrypt</td>
<td>18.2</td>
</tr>
<tr>
<td>aes2 postgres</td>
<td>tamper</td>
<td>17.8</td>
</tr>
<tr>
<td>htee postgres</td>
<td>encrypt</td>
<td>3.5</td>
</tr>
<tr>
<td>htee postgres</td>
<td>decrypt</td>
<td>75.4</td>
</tr>
<tr>
<td>htee postgres</td>
<td>tamper</td>
<td>58.8</td>
</tr>
<tr>
<td>htee console</td>
<td>encrypt</td>
<td>00.7</td>
</tr>
<tr>
<td>htee console</td>
<td>decrypt</td>
<td>81.9</td>
</tr>
</tbody>
</table>
The data presented in Table 3 shows the effect that the number of buckets and plaintext size has on the operation of the HTEE scheme. In all cases, both AES methods had equivalent performance times, between 15 and 18 seconds for encryption and decryption. The HTEE scheme had faster encryption times than AES, and the number of buckets only affected HTEE encryption performance marginally. The most distinguishing differences are found in the HTEE decryption and HTEE tamper detection tests. Due to the exhaustive search required for decryption, as the number of buckets increased the processing time increased. The cost per additional bucket decrypted is about 20 seconds. When processing the tampered data set, HTEE performance did not decrease as quickly for additional buckets. This is because the process was able to identify the tampered data in the first bucket processed. In these cases, each additional bucket added about 10 seconds to the processing time because half of the dataset was not tampered with and the full search process was required. For a dataset that is entirely modified, the identification of tampered records would be equal in performance across different bucket sizes.

<table>
<thead>
<tr>
<th>bucket size</th>
<th>Detailed Performance (time in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>aes1 postgres</td>
<td>encrypt</td>
</tr>
<tr>
<td></td>
<td>decrypt</td>
</tr>
<tr>
<td></td>
<td>tamper</td>
</tr>
<tr>
<td>aes2 postgres</td>
<td>encrypt</td>
</tr>
<tr>
<td></td>
<td>decrypt</td>
</tr>
<tr>
<td></td>
<td>tamper</td>
</tr>
<tr>
<td>htee postgres</td>
<td>encrypt</td>
</tr>
<tr>
<td></td>
<td>decrypt</td>
</tr>
<tr>
<td></td>
<td>tamper</td>
</tr>
<tr>
<td>htee console</td>
<td>encrypt</td>
</tr>
<tr>
<td></td>
<td>decrypt</td>
</tr>
</tbody>
</table>

The charts displayed in Figures 9 and 10 provide a graphical representation of the performance results. Figure 9 depicts the performance of the HTEE scheme versus the two AES test methods. It is clear that the AES methods provide consistent performance near seventeen seconds for each run. The HTEE scheme provides consistent fast performance for encryption at less than five seconds per run, but the processing time for decryption increases to over two minutes depending on the number of buckets processed.
Figure 9 - Performance comparison of AES vs. HTEE methods

The data shown in Figure 10 specifically focuses on the HTEE test results, showing the performance pattern graphically. The encryption operation is extremely efficient across all bucket sizes, because it is a straightforward hash/HMAC digest calculation. The decryption operation is significantly less efficient, averaging at a 21-fold increase in processing time over HTEE encryption. This is due to the exhaustive searches across possible bucket values for HMAC digest matches. Performance while processing the tampered dataset improved efficiency over the decryption operation, because of early exit logic used when no matching HMAC digest could be found.

Figure 10 - HTEE performance difference across bucket sizes
6.3. Performance Analysis

The performance results from testing indicate a four-fold decrease in encryption time and four-fold increase in decryption time over AES. This would be a reasonable tradeoff for some encryption heavy domains. The HTEE scheme also shows a performance improvement over the original HMAC encryption scheme [1], as shown in the following analysis. This analysis uses basic information about the original scheme's performance; a detailed summary is available in [2]. The performance of the HTEE scheme and the original scheme can be modeled and compared based on the algorithmic structure of the methods. The performance of the two schemes is generalized based on the number of HMAC operations required for encryption and decryption. Each HMAC operation includes two hash calculations, one inner and one outer. The hash and HMAC functions have a set number of bit operations which is not a concern here. The relative efficiency of the HTEE and original schemes can be found by treating the HMAC processing cost as a fixed value.

For the following analysis, let \( P_b \) be the number of buckets used, \( n \) be the plaintext value and \( S_b \) be the bucket size. For the HTEE scheme, the number of buckets can range from 1 through 6 based on the formula \( P_b = \text{floor}(\log_{1000}(n)) + 1 \) and the bucket size is fixed at 1,000. For the original HMAC encryption scheme, the number of buckets is fixed at two but the bucket size is variable. Ideally, the bucket size is equal to the square root of the maximum plaintext value, or \( S_b = n^{0.5} \) for a single value. The maximum value \( p \) that can be represented by these encryption schemes is related to bucket size and number of buckets as \( p < S_b^{P_b} \), if the buckets are of equal size as presented here.

The HTEE scheme’s encryption operation can be represented with \( P_b + 4 + P_b \) HMAC operations. The first \( P_b \) represents the HMAC operation to encrypt each bucket, the \( 4 \) is the number of HMAC operations required for element key transformation, and the second \( P_b \) represents the HMAC operations for bucket key transformation. For the decryption process, the number of hash operations required can be represented with \( S_b \cdot P_b + 4 + P_b \). In this case, the cost for element and bucket key transformation is unchanged, but the HMAC encryption cost is expanded to the possible bucket size. For very large numbers, the \( 4 \) can be disregarded, and the complexity can be summarized as approximately \( 2 \cdot \log_{1000}(n) \) for encryption and \( 1001 \cdot \log_{1000}(n) \) for decryption.

These performance expectations are compared against the original HMAC encryption scheme proposed in [1]. Based on the analysis of the original scheme presented in [2], the encryption and decryption operations are essentially equal in efficiency if processing a single plaintext value. The encryption operation recursively executes HMAC based on the value of the plaintext, and the decryption operation searches through all possible plaintext values for a match. When the maximum bucket ID and bucket size are equal, the efficiency
can be represented as \( \{S_b \times 2\} \). To generalize performance this value is used for both encryption and decryption. In this case the number of buckets \( \{P_b\} \) is fixed at two, one for the bucket ID and one for the remainder. For large numbers the complexity can be summarized as approximately \( \{2 \times n^{0.5}\} \) for encryption and decryption. The relative complexity of the HTEE scheme and the original HMAC encryption scheme are presented in Table 4.

**Table 4 - Complexity of HTEE and Original schemes**

<table>
<thead>
<tr>
<th>Encryption Scheme</th>
<th>Relative complexity (number of HMAC operations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTEE Encryption</td>
<td>( 2 \times \log_{1000} (n) ) Constant</td>
</tr>
<tr>
<td>HTEE Decryption</td>
<td>( 1001 \times \log_{1000} (n) ) Constant</td>
</tr>
<tr>
<td>Original Encryption</td>
<td>( 2 \times n^{0.5} ) Polynomial</td>
</tr>
<tr>
<td>Original Decryption</td>
<td>( 2 \times n^{0.5} ) Polynomial</td>
</tr>
</tbody>
</table>

Based on the relative complexity, the performance expectations for HTEE and the original scheme are compared in Table 5. Plaintext values ranging from 100 to \( 1 \times 10^{13} \) are modeled, and processing costs are calculated for both the HTEE and original schemes. The associated number of buckets and bucket size are displayed for clarity. The number of buckets used for HTEE is equal to \( \{\text{floor}(\log_{1000}(n)) + 1\} \), and the bucket size used for the original scheme is equal to \( \{n^{0.5}\} \). Encryption and decryption costs are displayed in bold.

**Table 5 - Performance comparison among HMAC encryption methods**

<table>
<thead>
<tr>
<th>Plaintext value</th>
<th>HTEE Scheme</th>
<th>Original Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bucket Size</td>
<td>Number Buckets</td>
</tr>
<tr>
<td>100</td>
<td>1,000</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>1,000</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>1,000</td>
<td>2</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000</td>
<td>4</td>
</tr>
<tr>
<td>10,000,000,000</td>
<td>1,000</td>
<td>4</td>
</tr>
<tr>
<td>100,000,000,000</td>
<td>1,000</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000,000</td>
<td>1,000</td>
<td>5</td>
</tr>
<tr>
<td>10,000,000,000,000</td>
<td>1,000</td>
<td>5</td>
</tr>
</tbody>
</table>
The performance improvements achieved with the HTEE scheme are significant over the original HMAC scheme for general encryption and for decrypting large numbers. Encryption is much faster with HTEE because it uses a direct HMAC calculation to build the ciphertext, rather than the recursive HMAC used by the original scheme. Decryption for HTEE is slower with small numbers (under one million) due to the fixed cost of the 1,000 bucket size. As plaintext values increase, the bucket size for the original scheme increases quickly making it much less efficient than the HTEE scheme which increases at a constant rate. This result is driven by the relationship of bucket size, number of buckets and plaintext value: \( p < S_b^{ph} \). The largest performance differences between the two schemes are found when processing large numbers, a result of increasing bucket sizes. In these cases HTEE has realistic decryption costs, but the original scheme has very inefficient and prohibitive decryption. One tradeoff for the larger number of buckets used by HTEE is an increase in the amount of ciphertext data stored. In the case of HTEE with six buckets, an 8 byte integer is encrypted into 168 bytes of base64 encoded HMAC digest data as ciphertext. The original scheme would represent this data in 56 bytes of base64 encoded data.

Performance testing verifies the improvement in processing time with the HTEE scheme over the original HMAC encryption method. As presented in [2], a test of the original scheme with 2,000 integer values less than or equal to \( 9 \times 10^8 \) using two buckets with size 50,000, encryption took 2 minutes and decryption took 3 minutes. These results are much slower than the HTEE performance times seen with all of the 20,000 integer datasets shown in Table 3. It would be prohibitive to encrypt integers up to \( 1 \times 10^{13} \) with the original scheme as modeled in Table 5. Based on the relative number of HMAC operations shown in Table 5 for the HTEE scheme and the original HMAC encryption method, the 20,000 record tests executed for HTEE would take a very long time using the original scheme.

The performance of the HTEE scheme used for this project can be summarized as follows. Compared against AES encryption methods for tamper detection, HTEE is more efficient on encryption and less efficient on decryption. The difference is approximately a factor of four with each operation; encryption is four times faster than AES and decryption is four times slower than AES. Compared against the original HMAC integer encryption scheme, the HTEE method is much more efficient for the decryption of large numbers and for general encryption. The relative complexity between the HTEE and original schemes is constant versus polynomial [16], which results in the improvement. These conclusions are based on an analysis of the relative number of HMAC operations required, and are verified against performance figures presented in [2]. A tradeoff with the HTEE scheme is the amount of ciphertext generated for large numbers of buckets. Using an input of 8 byte integer in the PostgreSQL environment, the AES encryption method produces 116 bytes of
base64 ciphertext data, and the HTEE scheme produces 168 bytes of base64 ciphertext data. Considering performance HTEE is preferred for problem domains where high performance encryption is required, but decryption performance is not a concern, and space is not a concern.

7. Conclusion

7.1. Overview of Results

The HTEE scheme provides a framework for tamper detection and encryption of integers in a database environment that can be useful in some applications. Benefits to the approach include the simplicity of a single-column confidentiality and integrity solution, trustworthy tamper detection based on a hash function, and efficient encryption speed. Drawbacks to the approach include inefficient decryption and increased volume of ciphertext.

The security analysis shows that the cryptographic strength of HTEE is based on the HMAC function and in turn the underlying hash function, SHA-1. Recent work suggests that HMAC is not affected by collision attacks against SHA-1 [4, 5]. Key recovery attacks are a threat to the HTEE scheme but these are still considered infeasible, and require a very large number of valid HMAC authentication codes [6, 7, 10]. Until a complete mathematical proof is generated, HTEE is considered not as secure as the AES encryption standard, and applications bound by regulatory or legal requirements should continue to use AES methods.

The HTEE scheme is distinguished by plaintext decomposition into multiple buckets and secret key transformation functions. The multiple bucket solution makes decryption feasible for large integers, and key transformation functions increase security through layering and provide tamper detection through unique related values. The scheme can detect changes between a stored ciphertext value and other data related to it such as a record’s primary key or hash digest value. The tamper detection feature is only provided on decryption, in order to be alerted to database tampering, the records must be decrypted.

The performance of the HTEE scheme is faster on encryption than AES, but slower on decryption. The differences are a factor of four in each case. For large numbers, the HTEE scheme is several orders of magnitude faster than the HMAC based encryption scheme it is based on [1, 2]. The HTEE scheme produces 44% more ciphertext data than an equivalent AES encryption scheme.

Applications for the HTEE scheme include areas where integer data is used, fast encryption speed is desired, slow decryption speed is not a significant concern, and tamper
detection is needed. An example of this would be auditing systems or the archival of financial transactions, such as bank or credit card activity. In these cases, a large number of records can be created on a daily basis, but the records might be infrequently referenced in the future. The HTEE method can support regular insertions into archive tables as opposed to a block encryption method that would require re-encryption of the entire data column. In a database that is write-only, or has little read access of encrypted records, HTEE can provide efficient tamper evident encryption as a supplementary protection for the database system. An example of this application could be storage of archival information by a third party, so the owner of the data can encrypt and protect data from tampering in addition to system level read controls implemented by the storage provider. In these situations, fast encryption is desirable and slow decryption is acceptable. If the stored records include financial transaction amounts or account numbers, these data could be encrypted with HTEE to ensure that the data has not changed since it was encrypted. If dollar amounts are processed, they must be multiplied by 100 first in order to capture the cents as part of the integer plaintext value.

### 7.2. Future Work

Some opportunities for future work related to this project and the HTEE scheme include support for expanded plaintext values and a rigorous security proof. The HTEE scheme improved the original HMAC encryption concept to make encryption of integers up to $9 \times 10^{17}$ feasible. However, the scheme is still limited to positive integer values because there is no way to encode negative or floating point values. A future improvement to the method could be a mechanism to process negative numbers, floating point numbers, and potentially ASCII-encoded text data.

This paper presented a conceptual argument for HTEE security based on existing work for HMAC security and key recovery. Based on the designed structure of HTEE, this provides a reasonable assurance of cryptographic strength because HMAC is the underlying function used, and it is widely considered to be a secure process. The security of HTEE is based on the HMAC function as a pseudo-random generator, both for key transformation and encryption. Future work can present a proof of the security for HTEE, which should focus on the random-generation capability of HMAC with the unique values used in the key transformation process.

### 8. References

[1] Dong Hyeok Lee; You Jin Song; Sung Min Lee; Taek Yong Nam; Jong Su Jang, "How to Construct a New Encryption Scheme Supporting Range Queries on Encrypted Database,"


URI: http://cseweb.ucsd.edu/users/mihir/papers/kmd5.pdf


URI: http://www.openauthentication.org/pdfs/Attacks%20on%20SHA-1.pdf


[11] Jongsung Kim; Alex Biryukov; Bart Preneel; and Seokhie Hong, “On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1”, 2006


URI: http://en.wikipedia.org/wiki/HMAC


URI: http://en.wikipedia.org/wiki/Birthday_attack


URI: http://www.koders.com/c/fidF9A73606BEE357A031F14689D03C089777847EFE.aspx

URI= http://base64.sourceforge.net/b64.c
URI= http://www.postgresql.org/docs/8.4/static/index.html
URI = http://doi.acm.org/10.1145/1103780.1103784
URI = http://doi.acm.org/10.1145/1103780.1103783
URI = http://doi.acm.org/10.1145/1412331.1412342
URI= http://en.wikipedia.org/wiki/Information_security
URI = http://cs.uccs.edu/~gsc/pub/master/bbaker/

Appendixes

Appendix A: Detailed Pseudocode

The following pseudocode highlights aspects of the HTEE design and implementation including the plaintext bucket decomposition, element key transformation, encryption function and decryption function.

Procedure 1: Bucket Decomposition.

The decompose procedure breaks a plaintext or ciphertext into bucket components of size 1,000. The input argument {value} is a positive integer.
Procedure decompose(value)
  If value is plaintext
    #plaintext decompose breaks input into buckets size 1000
    residual = value
    P = floor(log_{1000}(value)) + 1
    Loop while(P >= 0)
      #use modulus iteratively to decompose into buckets size 1000
      bucket = (residual - residual mod 1000^P) / 1000^P
      residual = residual mod 1000^P
      P = P - 1
    EndLoop
    Return list of bucket
  Else
    #ciphertext decompose breaks input into digests of 20 bytes
    #with sha1, bucket digests are 20 bytes, 28 bytes base64
    Parse value into bucket digests
    Return list of digests
  EndIf

Procedure 2: Element Key Transformation.

The element procedure performs the unique value based key transformation
using the HMAC function and original secret key. The input argument \{K^O\} is the
binary format original secret key, and the argument \{unique\} is a unique text value
or hash digest related to the plaintext data.

Procedure element(K^O, unique)
  #hash the unique value to seed the HMAC iterations
  temp = Sha1(unique)
  #iteration 1, HMAC using original key
  temp = HMAC(K^O, temp)
  #HMAC result becomes first 20 bytes of key
  K^E = temp
  #iteration 2, HMAC using original key
  temp = HMAC(K^O, temp)
  #HMAC result becomes second 20 bytes of key
  K^E = K^E + temp
  #iteration 3, HMAC using original key
  temp = HMAC(K^O, temp)
  #HMAC result becomes third 20 bytes of key
  K^E = K^E + temp
  #iteration 4, HMAC using original key
  temp = HMAC(K^E, temp)
  #HMAC result becomes last 4 bytes of key
  K^E = truncate(K^E + temp)
  Return K^E
End.
Procedure 3: Encryption Operation.

The HTEE procedure performs encryption on plaintext bucket values using the HMAC function and the element key. The procedures “decompose” and “element” are referenced in this pseudocode. The bucket key transformation is included as part of this procedure. The input argument \{plaintext\} is a positive integer value less than $9 \times 10^{17}$, the argument \{unique\} is a unique text value related to the plaintext data, and the argument \{K^O\} is a secret key in binary format.

```
Procedure HTEE(plaintext, unique, K^O)
Begin
    #find the size of the number
    buckets = floor(log_{1000}(plaintext)) + 1

    #transform key for this plaintext element
    K^E = element(K^O, unique)

    #bucket key starts as element key
    K^B = K^E

    #loop through buckets for HMAC operation
    For j=1 to buckets
        #find the bucket value and HMAC it
        b = decompose(plaintext)
        c = HMAC(b, K^B)

        #transform key for bucket value
        #prepend the HMAC result to the key, truncate to key size
        K^B = HMAC(c, K^O) + K^B

        #accumulate ciphertext from all buckets
        ciphertext = ciphertext + c
    Endfor

    return ciphertext
End.
```

Procedure 4: Decryption Operation.

The HTEE procedure performs decryption on ciphertext bucket values using an exhaustive search of plaintext values with the HMAC function and the element key. The procedures “decompose” and “element” are referenced in this pseudocode. The bucket key transformation is included as part of this procedure. The input argument \{ciphertext\} is a base64 encoded string, in a multiple of 28 bytes, the argument \{unique\} is a unique text value related to the plaintext data, and the argument \{K^O\} is a secret key in binary format.
Appendix B: Add-on Compilation and Installation

This appendix presents notes to support compilation of the PostgreSQL database add-on as used in this project. The below instructions include support for compilation or direct use of delivered DLL files. Variations in compilation or server environment can require additional modifications to compile and execute the database add-on. The changes listed below include standard modifications to the compilation environment and workaround modifications to PostgreSQL header files. Future versions of the PostgreSQL server distribution may not require the listed modifications to header files. Source code for this project is available at http://cs.uccs.edu/~gsc/pub/master/bbaker/src/.

This project was compiled and tested on:
- Windows XP
- Microsoft Visual C++ 2008 Express Edition
- PostgreSQL 8.3.8 (server and include files)
- PostgreSQL 8.4.1 (server and include files)

Step 1: Installation (always required)
- Install PostgreSQL 8.3 or 8.4 server, including development header files
- Note the installation path of PostgreSQL
  - The typical Windows installation path is:
    "C:\Program Files\PostgreSQL\8.3\", depending on server version.
  - In the following sections this is referred to as: %PGPATH%
- For compilation and creation of the add-on DLL:
- Install Microsoft Visual C++ 2008 Express Edition
  - Without compilation, if using the delivered DLL files:
    - Install Microsoft Visual C++ 2008 Redistributable Package (x86)

**Step 2: Configuration of compiler (only required if compiling add-on)**
- Create empty project
- Add HTEE source files to project
- Open project properties and make these settings:
  - General: Configuration type:
    - dynamic library (dll)
  - C/C++ General: Additional include directories:
    - (PGPATH with PostgreSQL installation path)
      - "%PGPATH%/include\server\port\win32"
      - "%PGPATH%/include"
      - "%PGPATH%/include\server"
  - C/C++ Advanced: Compile as:
    - C code
  - Linker general: Additional library directories:
    - (PGPATH with PostgreSQL installation path)
    - "%PGPATH%/lib"
  - Linker: Input: Additional dependencies:
    - postgres.lib

**Step 3: Modifications to PostgreSQL headers (only required if compiling add-on)**
- Due to differing system settings or compilers, the delivered PostgreSQL header files don't compile with Visual C++ 2008 and some updates are required.
- These fixes will resolve compilation or run-time errors. They may not be needed on all systems.
- In the file: %PGPATH%/include/pg_config.h
  - Change "#define ENABLE_NLS" from 1 to 0
- In the file: %PGPATH%/include/pg_config_os.h
  - Comment out the struct definitons for "itimerval" and "timezone"
- In the file: %PGPATH%/include\server\c.h
  - Comment out "#include <libintl.h>"
- In the file: %PGPATH%/include\pg_config_os.h
  - Change "#define PGDLLIMPORT __declspec (dllimport)" to "#define PGDLLIMPORT __declspec (dllexport)"
- In the file: %PGPATH%/include\server\utils\elog.h
Step 4: Installation and testing of add-on (always required)
- After a successful compile, copy the created dll file to the "$PGPATH\lib" directory
- Connect to the PostgreSQL server, through the command line or a utility like pgAdmin
- Copy the "HTEE.conf" file and "key.txt" file to the PostgreSQL data directory (often "$PGPATH\data")
- Execute these SQL commands, replacing "HTEE_pgsql" with the filename of the dll:
  o CREATE FUNCTION htee_enc(int8, text) RETURNS text
  AS '$libdir/HTEE_pgsql', 'htee_enc'
  LANGUAGE C CALLED ON NULL INPUT;
  o CREATE FUNCTION htee_dec(text,text) RETURNS int8
  AS '$libdir/HTEE_pgsql', 'htee_dec'
  LANGUAGE C CALLED ON NULL INPUT;
- Create a basic table with two columns: a text primary key "keyval" and a bigint data field "dataval".
- Test with the following SQL:
  o SELECT keyval, dataval, htee_enc(dataval,keyval),
  htee_dec(htee_enc(dataval,keyval),keyval) FROM test;
- The original plaintext, encrypted ciphertext and decrypted plaintext should display.

Appendix C: SQL for Testing

This appendix presents SQL statements used during test runs of the raw AES, AES with tamper detection, and HTEE encryption methods. In the test database schema, the {htee_test1} table contains plaintext data and ciphertext data for each encryption method in different columns {data, cipher_aes1, cipher_aes2, cipher_htee}, and primary key {id}. The secret key is stored in the {keys} table for convenience. The HTEE specific functions are {htee_enc} for encryption, and {htee_dec} for decryption. All other functions are delivered PostgreSQL functions, including {encode, decode, pgp_sym_encrypt, pgp_sym_decrypt, cast, strpos and strlen}. 

41
Encryption operation:

- AES1 method: Raw AES encryption with no tamper detection
  
  ```sql
  UPDATE htee_test1 SET cipher_aes1 = 
  encode(pgp_sym_encrypt(cast(data as text),
  (SELECT key FROM keys WHERE id=1),
  'cipher-algo=aes128'),'base64');
  ```

- AES2 method: AES encryption with tamper detection through unique value concatenation
  
  ```sql
  UPDATE htee_test1 SET cipher_aes2 = 
  encode(pgp_sym_encrypt(id||';'||cast(data as text),
  (SELECT key FROM keys WHERE id=1),
  'cipher-algo=aes128'),'base64');
  ```

- HTEE method: HTEE encryption with unique value for key transformation
  
  ```sql
  UPDATE htee_test1 SET cipher_htee = 
  htee_enc(data,cast(id as text));
  ```

Decryption Operation:

- When data has been tampered with, the AES2 and HTEE methods will produce a decryption result of -1, indicating the tamper.

- AES1 method: Raw AES encryption with no tamper detection
  
  ```sql
  UPDATE htee_test1 SET dec_aes1 = 
  cast(pgp_sym_decrypt(decode("cipher_aes1","base64"),
  (SELECT key FROM keys WHERE id=1),
  'cipher-algo=aes128')as bigint);
  ```

- AES2 method: AES encryption with tamper detection through unique value concatenation
  
  ```sql
  UPDATE htee_test1 SET temp_aes2 = 
  pgp_sym_decrypt(decode("cipher_aes2","base64"),
  (SELECT key FROM keys WHERE id=1), 'cipher-algo=aes128');
  
  UPDATE htee_test1 set dec_aes2 = cast(substr(temp_aes2,
  strpos(temp_aes2,';')+1,length(temp_aes2)) as bigint)
  
  = cast (id as text);
  
  UPDATE htee_test1 SET dec_aes2 = -1
  
  WHERE substr(temp_aes2,0,strpos(temp_aes2,';'))
  <> cast (id as text);
  ```

- HTEE method: HTEE encryption with unique value for key transformation.
  
  ```sql
  UPDATE htee_test1 SET dec_htee = 
  htee_dec(cipher_htee, cast(id as text));
  ```