Protecting Secret Data from Insider Attacks

David Dagon, Wenke Lee, and Richard Lipton

Georgia Institute of Technology
{dagon, wenke, rjl}@cc.gatech.edu

Abstract. We consider defenses against confidentiality and integrity attacks on data following break-ins, or so-called intrusion resistant storage technologies. We investigate the problem of protecting secret data, assuming an attacker is inside a target network or has compromised a system.

We give a definition of the problem area, and propose a solution, VAST, that uses large, structured files to improve the secure storage of valuable or secret data. Each secret has its multiple shares randomly distributed in an extremely large file. Random decoy shares and the lack of usable identification information prevent selective copying or analysis of the file. No single part of the file yields useful information in isolation from the rest. The file's size and structure therefore present an enormous additional hurdle to attackers attempting to transfer, steal or analyze the data. The system also has the remarkable property of healing itself after malicious corruption, thereby preserving both the confidentiality and integrity of the data.

1 Introduction

Security technologies have traditionally focused on perimeter defenses. By itself, this approach creates what has been called a lobster-model of security, or "a sort of crunchy shell around a soft, chewy center" [Che90]. If an attacker manages to get into the network, it becomes very difficult to detect or prevent further security compromises.

This has prompted the development of secure storage techniques that resist against successful attacks. This paper studies the problem of protecting secret data under the assumption that an attacker has already broken through the network perimeter (or is an "insider"). We give a formal definition of the problem, and present one solution called VAST. The key idea is to distribute secret data in an extremely large storage system without exploitable identification information. Our VAST storage system is orthogonal and complimentary to existing data protection techniques, such as encryption, in that it makes attacks much more difficult to succeed.

In this paper, we describe the design rationales, data structures and algorithms. We also describe an implementation of such a system, to demonstrate acceptable normal use. Specifically, we make the following contributions:

Definition of Secure Storage Problem. We formally describe the problem of secure storage of secrets in Section 3.1. We describe an abstract data type that is a large storage table composed of records. Operations include initialization, insertion and deletion. We also describe security properties that the table and operations must guarantee. This

A.S. Patrick and M. Yung (Eds.): FC 2005, LNCS 3570, pp. 16–30, 2005.

general description of the problem formalizes intrusion resistant systems, and encourages further research into this general problem area.

Storage Scheme for Secret Data. Based on the abstract data type, we propose the VAST storage system, which uses extremely large (e.g., terabyte-sized) files to store secret information. In VAST, a secret is broken into shares distributed over a large file, so that no single portion of the file holds recoverable information.

2 Related Work

VAST of course fits into the larger field of fault-tolerant systems generally, and intrusion-tolerant systems specifically. There has been a considerable amount of work on tolerant and dependable file storage. Many works have used secret sharing as part of a resilient data storage system [WBS+00, LAV01]. Of particular relevance is [FDP91, FDR92], and Rabin's work in [Rab89], which all used secret sharing as an information dispersal technique for security and redundancy. Our work is in a similar vein, but seeks to leverage tradeoffs between disk I/O speeds and memory on *local* file stores, without the need to distribute shares among hosts.

Many other intrusion resistant systems have used *fragmentation-and-scattering*, a technique similar to VAST's hashing store of secret shares. In [DFF+88], the SATURNE research project describe the fragmentation-and-scattering scheme. Stored data was cut into insignificant fragments, and replicated over a network. The distribution of the fragments obliged attackers to compromise numerous resources before they could read, modify or destroy sensitive data. Instead of distributing resources over a network, VAST keeps all fragmented data in a single file, albeit usually spread over several drives.

The tremendous time difference between memory and drive I/O has motivated work in complexity analysis [AKL93, AV87]. The general goal of these works is to describe a lower bound on algorithms and demonstrate a minimal number of I/O operations. VAST works in the opposite direction, and seeks to maximize the number of required I/O operations to slow attackers.

Components of VAST were inspired by other results. For example, the large table in VAST is similar in principle to the solution in [Mau92], where a short (weak) key and a long plaintext were kept secure by using a publicly-accessible string of random bits whose length greatly exceeded that of the plain text. In [CFIJ99], the authors created a very similar model for memory storage, and generally described how to create a storage system that can forget any secret. Their solution assumed the existence of a small and fixed storage area that the adversary cannot read, which differs from VAST's large, unfixed, and readable storage tables.

Other areas of research have used techniques similar to VAST. VAST's distribution of shares over a table has a superficial resemblance to data hiding [KP00]. VAST's ability to recover and heal corrupted messages also resembles Byzantine storage systems [MR98], or even the larger field of error correction codes. VAST combines existing approaches in a new and interesting way.

3 Designing Large Files for Valuable Data

Below, we describe an abstract secure storage problem, and suggest relevant design considerations for any solution. We then propose the VAST storage system, and detail its operation.

3.1 Secure Storage Problem Statement

For this paper, we address the following specific scenario: Assuming an attacker has penetrated a storage system, what reasonable measures help prevent the compromise of stored secret data through brute-force analysis, such as key cracking, dictionary password guessing, and similar attacks?

We formally describe the secure storage of data in large tables as follows. A large table T has parameters (n, r, m, d, K). The table is used to store n records of r bits. The table itself is m bits in size, where $m \geq nr$, and usually $m \gg nr$. The value d determines a fraction of the table, $0 < d \leq 1$. The value K, described below, is a threshold used to measure security properties. The table supports the following operations.

1. **Initialize.** An $init()$ function iteratively initializes each of the n records in T.
2. **Add.** An $add()$ operation inserts data into the table.
3. **Delete.** A $delete()$ operation removes entries from the table.
4. **Find.** A $find()$ operation retrieves information from the table.

The security property of the table is the following statement. Suppose we initialize the table and then perform a series of insertion operations. Next, suppose we use only dm bits from the table. Given a value x, and using only dm bits, the probability one can can correctly compute $find(x)$ is at most 2^{-K}. In other words, if dm bits are stolen or analyzed, there's only a small chance that x can be recovered from the exposed portion of the table. We can also state a stronger security property for the table, so that it also provides semantic security. Again assuming only dm bits are used, the semantic security property holds that one cannot compute $find(x)$ correctly, and further cannot obtain one bit of x with any advantage over $\frac{1}{2} + 2^{-K}$.

It is not obvious that one can create a table with these properties. Reasoning about the problem points to one possible solution. To start, we know that our overall goal is to increase K, which minimizes the probability of a successful attack. One strategy to accomplish this is to encrypt the data, x, inserted into the table, since this makes linear scans of the table much more difficult, and forces the attackers to perform brute-force attacks on the encryption scheme. A second strategy is to not only increase m, but also to distribute x in such a way that $d \approx 1$ before recovering x becomes possible. In other words, we should store data in a large table such that analyzing a small d fraction of the table cannot yield x.

An additional, practical benefit derives from using a large table size, m. If the table is large, and x is stored such that d must be near 1, then in practical terms this means analyzing the table's m bits will require enormous resources. We know, for example, that I/O access is extremely slow compared to memory access [AKL93]. We therefore should design our table with a goal opposite of Vitter's work minimizing I/O operations

in algorithms [AV87]. Instead, we wish to *maximize* the I/O operations (and therefore, the time) required for analysis.

The above discussion suggests making the table size large. One consequence is that an attack will take more time to succeed. With I/O operations *several* orders of magnitude slower than memory access [HP03], this means analysis will require repeated disk access.

3.2 Design Considerations for Secure Data Storage Problems

In most attacks on data confidentiality and integrity, the attacker first needs to get hold of the target data, usually by copying it offsite. In this attack set up stage, time is proportional to the size of data. For example, if the attacker needs to transfer data on a link with a capacity of C data units per unit time, then the time it takes to transfer data with size D will be $T = \frac{D}{C}$. If the target data is actually small in size, we better protect the data by dispersing it in a large storage file without any "exploitable" identification information. This will force the attacker to process the entire large storage to recover the target information. If the table size m is tera-scale, the time needed to steal the file is potentially prohibitive.

In order to slow the attack, we need to force it to carry out more operations. For attacks on confidentiality and integrity, a simple protection scheme is to fragment the data and distribute the shares throughout the large file. Thus, for each attack (trial) that involves locating shares and guessing (brute-force analyzing the data), instead of spending time T for one target, it now must spend time kT if k fragments are needed to reconstruct the data.

3.3 The VAST Storage System

We now describe the design of our large file scheme, using a credit card database storage system as a motivating example. User financial records are stored in a file, and retrieved using keys, passwords or PINs that hash to appropriate table entries. (Without significant modification, the system could be used in almost any password-based authentication system.) A readable metadata index file stores the relevant information for each user, including user name, *u*, and salts s_1, s_2, ..., s_k, each a random number. The metadata identification (or user identity) file does not need to be read-protected because it contains no secret. (In practice, of course, one might elect to restrict access to this file as well; however, our analysis presumes it has been accessed by an attacker.)

The data storage file is a very large table T with m entries in which multiple shares of data are randomly distributed. There are no empty entries because the table is initially filled with random bit strings that look like valid shares.

We next study the data structures and algorithms for the large table file. The main design goals are:

Functional. From the functionality point of view, the table must store financial information reliably so that the data is retrievable only when a proper key is presented. This corresponds to the $add()$ and $find()$ operations noted in section 3.1.

Secure. From the data security point of view, the design objective is to make it very difficult and slow for an attacker to steal the large information file and extract the infor-

mation using brute-force key guessing or dictionary attacks. That is, it costs the attacker maximally (in time, or other resources) with each guess. This corresponds to the security principle noted in section 3.1.

Below, we discuss how to achieve these goals.

Storing Unguessable Shares of Random. In order to force the attacker to read all shares with each guess, VAST is based on secret sharing [Sha70]. Financial data for each user is stored under their unique name, u, in a large table. The data is accessed through the use of a key, key, and k random salts, $s_1, s_2, \ldots, s_k$.

To add a user and data into the system (the $add()$ operation in Section 3.1), we first take the user's financial information (e.g., a credit card number) M, and add any needed padding to match the length of X_2, a large random number selected for each insertion of M. We will use X_1 to refer to the padded information M. Together, X_1 and X_2 may be considered as a message and Vernam's one-time pad [Bis03]. As will be seen below, portions of this cipher scheme are stored in the table. We selected a one-time pad because its provable security was attractive, and helps partially address problems found in hash-storage schemes, such as dictionary attacks on weak passwords. The use of the pad also avoids problems associated with storing message-derived hashes in the metadata table, e.g., theft of hashes, and offline guessing attacks against messages with similar structures, such as credit cards. (We discuss an attack model below.)

X_1 and X_2 are of equal length, on the order of 128 to 160 bits or more. The numbers are XOR'd together to produce a third value, $X = X_1 \oplus X_2$. The random number X_2 is then appended to the user's entry in the identity file, along with user name u and a set of salts, $\{s_1, \ldots, s_{k'}, \ldots, s_k\}$, each a unique random number.

Instead of storing the padded message X_1 in the table, we first encrypt it with a symmetric encryption operation, $E_{key}(X_1)$. (Any symmetric encryption system can be used.) In addition to improving security, the encryption step also makes it easier to generate convincing initial (random) values for unused portions of the table.

Then, applying Shamir's secret sharing scheme [Sha70], two random polynomials are constructed to secret share $E_{key}(X_1)$ (the encrypted message) and X (the cipher text):

$$f_1(x) = E(X_1) + \sum_{j=1}^{k'-1} a_j x^j \pmod q, \quad f_2(x) = X + \sum_{j=1}^{k'-1} b_j x^j \pmod q \quad (1)$$

We select q, a large prime number (greater than k, X_1 and X_2), and store it in the metadata file. The coefficients a_j (and likewise b_j) $j = 1, 1, \ldots, k' - 1$, are random, independent numbers in the range of $[0, q-1]$. We use $k' \leq k$ to provide collision tolerance because k' shares are sufficient to reconstruct the secret. Thus, for k shares, the threshold of k' shares must be present to recover the secret. For each $i = 1, 2, \ldots, k$, we store both $f_1(i)$ and $f_2(i)$ in the same table entry at $H(key||s_i) \bmod m$. These shares look just like any other random numbers in the range $[0, q-1]$. Therefore, at initialization (the $init()$ operation in Section 3.1), the table is filled with random numbers in the range of $[0, q-1]$. After the shares are inserted in the table, the coefficients of the two polynomials (Equations (1)) are discarded. Figure 1 provides an overview of the process.

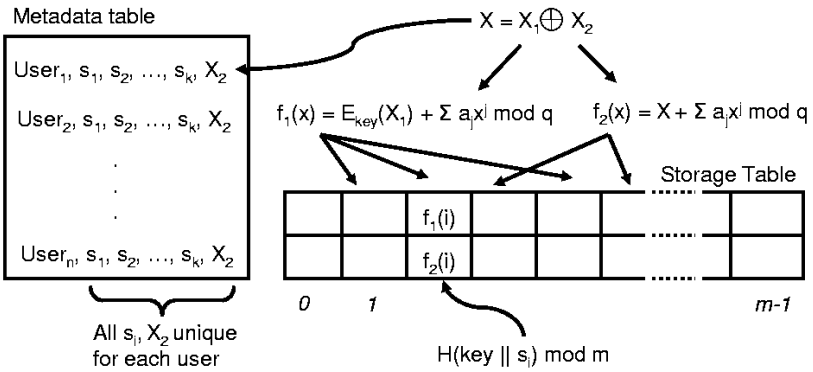


Fig. 1. *Overview of VAST System* Information or a message, X_1, is $\oplus$-combined with a random number X_2 to form X. The random number X_2 is stored in a metadata table under the appropriate user's entry, along with random salts, $s_1, s_2, \ldots, s_k$, unique for each user. The values $E_{key}(X_1)$ and X are Shamir-shared to derive k shares, f_1 and f_2. Each $f_1(i)$ and $f_2(i)$, which are stored in the large table, based on a hash of the key and salt, at table entry $H(key||s_i) mod\ m$

To retrieve information for a user u (the $find()$ operation in Section 3.1), we interact with the user to obtain the key, password or PIN, called key', and look up the salts in the metadata file. Then, we retrieve the shares $f'_1(i)$ and $f'_2(i)$ in the table entry $H(key'||s_i) \bmod m$, for each $i = 1, 2, \ldots, k$. Given k' shares, say $i = 1, 2, \ldots, k'$, the polynomial f_1 (and likewise f_2) can be reconstructed using Lagrange interpolation:

$$f'_1(x) = \sum_{i=1}^{k'} f'_1(i) \prod_{1 \le j \le k', j \ne i} \frac{x-j}{i-j} \pmod q \tag{2}$$

Thus, X'_1 (and likewise X') can be computed:

$$X'_1 = f'_1(0) = \textstyle\sum_{i=1}^{k'} c_i f'_1(i) \pmod q, \quad \text{where } c_i = \prod_{1 \le j \le k', j \ne i} \frac{j}{j-i} \tag{3}$$

We then perform decryption, $X'_1 = E^{-1}_{key}(E_{key}(X'_1))$. If $X'_1 \oplus X' = X_2$ (the value stored with u in the metadata file), then the key was valid, and the correct message X_1 was recovered. If $X'_1 \oplus X' \ne X_2$, this may be due to collisions (i.e., some shares overwritten by the shares of another user), and another k' shares can be used to compute X'_1 and X' as in Equation (3). In the worst case, one needs to try $\binom{k}{k'}$ times before the key is validated. However, since collisions are very rare, the probability of success (in validating a valid key) with the first k' shares is very high.

Suppose an incorrect key key' is supplied. Then k' incorrect shares $f'_1(i)$ and $f'_2(i)$, $i = 1, 2, \ldots, k'$ are read to construct X'_1 and X'. The chance of $X'_1 \oplus X' = X_2$, and thus validating the incorrect key' , is very small, 2^{-128} if X is a 128-bit random. This is because for the X' value computed from the shares, X'_1 must happen to be exactly $X' \oplus X_2$, which in turn requires that one share, say the k'th share, for X'_1, must be $X \oplus X'_2 - \sum_{i=1}^{k'-1} c_i f'_1 i \pmod p$, a 2^{-128} chance. Thus, VAST meets the security property for storage tables stated in Section 3.1.

An attacker may attempt to search for the shadow keys in T, since every data element has at least k' shares in the table. But searching for the correct k' elements in m is difficult, on the order of $\binom{m}{k} \geq (\frac{m}{k})^k$, where m is enormous. (Recall, the table is tera-scale, often with 2^{40} or more entries, all initially random.) The attacker's best strategy is key guessing, since the search space is much smaller. Even if 8 character keys are composed from 95 possible characters, it is easier to guess out of $95^8 \leq 2^{56}$ combinations, compared, say, to $\binom{2^{40}}{8} \geq 2^{296}$, for $k = 8, m = 2^{40}$. So, the attacker can only perform key guessing.

Now consider an attacker attempting to guess the key of user u to retrieve the financial data, X_1. If she can precompute the shares of X_1 and X, then for the guessed key *key'*, she might just check the shares in one entry, say $H(key'||s_1) \bmod m$, (or up to $k - k'$ entries) to learn that *key'* is incorrect. However, we can show that this is not possible. First, although she can read the identification file and hence the random X_2, she cannot figure out the values of message X_1 and cipher text X because encryption using one-time pad offers perfect secrecy. Furthermore, the coefficients of the polynomials in Equations (1) are random and are discarded. Therefore, there is no way the attacker can precompute the shares. So, she has to read the shares from the table. If she reads fewer than k' shares, according to Equation (3), she will not be able to compute X_1' (and likewise X'). And without X_1' and X', she cannot test if $X_1' \oplus X_2 = X'$ to know if *key'* is correct. Based on the above analysis, and the strengths of the basic cryptographic primitives of one-time pad and secret sharing, we have the following claim:

Property 1. In order to retrieve a user's information or just to learn that the key is incorrect, at least k' table entries must be read.

When collisions occur, and enough of the salts still point to valid shares, the system has detected some corruption of the table. In other words, some of the shares are invalid, but enough are still present to recover X_1 under the Shamir secret sharing scheme. This could be due to collisions as other data is added to the table, or because of malicious corruption of the file. In either case, if X_1 has been retrieved using only $k' < k$ salts, new random salts are generated, and written to the metadata file. The data is then rehashed and written to the table. This way, the table "heals" itself and corrects corruption detected during reads. Thus, when data collides with other entries, we eventually detect this problem, and relocate the shares. This movement may cause other collisions, but the chance is small. Eventually a steady state is obtained, and no user's shares collide with shares of any other. Section 4 discusses the reliability of this system, and the small probability of collisions occurring.

In order to completely corrupt a secret stored in the table, at least $k - k' + 1$ entries must be overwritten. The chance of this occurring with random writes is extremely small, on the order of $\frac{k-k'+1}{m}$, where m is enormous. Section 4 provides a complete analysis of the reliability of the system. However, if any legitimate read access occurs prior to all $k-k'+1$ collisions, the corruption will be detected and repaired. (Recall that only k' shares must be valid, so $k - k'$ corrupted shares can be detected and corrected in the course of legitimate use.) This property allows us to assert the following claim:

Property 2. Since reads from the table reveal any collisions, allowing for repair of the data's integrity, data is destroyed only if $k - k' + 1$ shares are corrupted between legitimate access attempts.

This is an important property for storage systems, since attackers unable to recover data from the file may nonetheless maliciously write bad information, in order to corrupt the file for normal use. (For example, they might randomly write zeros to the table.) With a large tera-scale file, however, successfully deleting all information would take an enormous number of writes, and may risk detection by other orthogonal detection systems.

The size of the financial data, M, stored as X_1 using the above scheme is of course limited [CSGV81]. We've used credit card information as a motivating example. However, there are many ways we can extend our scheme to store arbitrarily large files. One simple scheme is to treat each encrypted block of the whole encrypted message as an M of user i. In other words, we could make as many users are there are blocks, so that a large M is distributed or chained over many users.

No doubt other variations are possible. One can be creative about using pointers, indices, or set orders to store even large amounts of data. Therefore, while credit card number storage provides a real-world motivation for our work, our scheme can be extended to provide more general support for a wide range of applications. Tera-scale drives are now affordable, and we encourage others to examine how fragmentation-and-scattering schemes can be improved with large data stores.

Table Tiers. We also briefly note a possible variation of VAST using table tiers to efficiently use limited resources. While tera-scale drive storage is inexpensive, greater reliability may be obtained by dividing a quantity of storage into separate independent VAST tables.

Recall the important design goal of providing *reliable* storage for sensitive information. As will be discussed in 4.2, there is a small chance that collisions may occur when inserting new shares into a table. So, in addition to using a lower threshold for validating retrieved information, $k' \leq k$, one can simply make additional tables, each holding the same user information, but distributed with independent sets of salts. Thus, a 4-terabyte storage system can be broken into 4 1-terabyte storage systems, each with an independent chance of failure.

Using separate *independent* sets of salts over many separate tables is analogous to the practice of using drive backups from different manufacturers and models, in order to ensure that the hardware failure rates are truly independent. So, by adding tiers of tables, one can reduce an already small chance of failure into an infinitesimal risk.

4 Security Analysis

By storing secret data in a large, structured file, attackers are forced to copy and analyze the entire terabyte-sized file as a whole. No single portion of the file yields useful information in isolation. Below, we evaluate the improved security provided by VAST, and the reliability of the system.

4.1 Cost of Brute-Force Attacks

Below, we analyze the solutions VAST provides, namely (a) reliable and efficient retrieval of stored secrets, and (b) greater defense against key-cracking attacks.

Attacks in General. Broadly, attacks on storage files fall into two categories: on-line attacks and off-line analysis [PM99, Bis03]. The on-line analysis of keys is difficult in VAST for several reasons. First, scanning the hash file in a linear fashion does not provide the attacker with any information about which entries are valid hash stores. (Recall that unused entries are initialized with random bits, and data is stored in encrypted shares, which also appear random.) Interestingly, all of the k Shamir secret keys are present in the same file; however, the attacker has $\binom{m}{k}$ possible combinations. Recall that m is enormous, say in the range of 2^{40}, and k is not negligible, say in the range of 8-10. So $\binom{m}{k} \geq (\frac{2^{40}}{8})^8 \geq 2^{296}$, and the presence of all the shares on the table T does not help the attacker more than guessing.

Since sequential or adjacent shares on disk may be read more quickly than shares distributed on random parts of the drive, an attacker may attempt to precompute numerous hashes for key guesses, and upload the sorted precomputed indices. That is, an attacker might compute, using a dictionary D, with P permutations per word, some $\{|D| \cdot P \cdot nk\}$ hashes offline, and sort them by index value to improve drive access times, since many shares for many guesses will be adjacent, or at least within the same logical block on disk. (Recall, for example, that drive reads from adjacent locations on disk may be faster that reads from non-adjacent tracks and sectors [HP03].) However, if the VAST system is properly bandwidth limited, the attacker will find this slow going as well. The minimal space needed to request a single share is 8 bytes. Assuming a dictionary of just ten thousand words is used, with only a hundred permutations per word, the attacker would have to upload approximately 8 megs for each user *and* each salt. Because VAST systems are deployed on low-bandwidth links, this could potentially take a long time, and could easily be detected. Even if the attacker somehow uploaded the precomputed indices, they still have to obtain the k shares and find if any k' subset solves a polynomial to recover X_1 and X_2.

Without sufficient resources on-line, an attacker's preferred strategy would be to transfer the hash file for off-line for analysis. Assuming an attacker somehow transfers a tera-scale file offsite for analysis, the size of the file presents a second hurdle: repeated I/O operations.

Disk access takes on the order of 5 to 20 milliseconds, compared to 50 to 100 nanoseconds for DRAM. While drives are getting faster, they are on average 100,000 times slower than DRAM by most estimates [HP03], and are expected to remain relatively slow [Pat94].

Given this, Anderson's formula [Bis03] can be used to estimate the time it would take to check N possible keys, with P probability of success (of one key guess) and G guesses performed in one time unit: $T = \frac{N}{PG}$. To perform an exhaustive key space search, an attacker might load some of the hash file into memory, m', while the bulk of it, $m - m'$, must remain on disk. For those key guesses that hash to a memory store, the attacker would enjoy a fast lookup rate on par with existing cracking tools. But most of the time, the attacker would have to read from disk. Since VAST's indexing schema

uses hash operations that provide uniform dispersion, the ratio of memory to disk is applied to the rates for drive and memory access. We assume that the time required for a disk-bound validation operation is a factor of L of the time for memory-bound operation, and let $r = \frac{m'}{m}$. We can then modify the guess rate G in Anderson's formula to reflect the rate for disk access, so it becomes $G(r + (1 - r)L)$. Since k' shares must be read to validate a guessed key, the guess rate is further reduced to $\frac{G(r+(1-r)L)}{k'}$. We thus have the following claim:

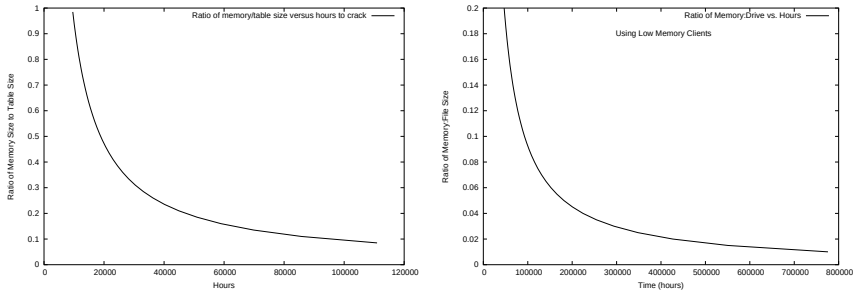
Property 3. In the VAST system, the time taken to successfully guess a key (with probability P) is:

$$T = k' \frac{N}{PG(r + (1 - r)L)} \tag{4}$$

In this light, existing encrypted file schemes are just a special case of the VAST system with $r = 1$ and $k' = 1$, and a much smaller m. Our objective is to make T as high as possible. If we make the table very large, r is close to zero, then Equation (4) is close to $T = k' \frac{N}{PGL}$. This means then the deciding factor is L, or the time required for disk access.

Our implementation of a single-CPU cracker resulted in a rate for memory-bound operations of just over 108,000 hash operations per second, while the disk-bound guessing yielded approximately 238 hash operations per second. No doubt, different hardware will produce different results. But on the whole, systems designers note that disk access is at least 100,000 times slower than accessing memory [HP03], i.e., $L = \frac{1}{100{,}000}$, so one might expect the ratio of L to improve only slightly [Pat94].

Using the modified Anderson's formula, we can estimate progress on a single machine, making the conservative assumption of a key alphabet of 95 printable characters,



(a) Guess Rate (b) Low Memory Client Guess Rate

Fig. 2. Figure (a) shows how the ratio of memory to table size affects guess rates for key cracking. The graph assumes 6 character keys selected from 95 printable characters, and 5 salts per user, and $m = 2^{40}$ entries. Reasonable progress is only possible when memory size is large enough to hold the entire table. Figure (b) shows the guess rate when low-memory clients are used, effectively zooming in on a portion of figure (a). With less memory, the guess rate is consistently slow. Administrators can force attackers into a low-performance portion of the curve just by adding inexpensive additional drives

merely 6 character keys, and only five salts per data item. Figure 2(a) plots the time it takes to guess a key, as a function of the ratio of memory to disk size. If one has a 1:1 memory disk ratio (i.e., a terabyte of memory, approximately $1.6 million [FM03]), the cracking time still requires over 9,500 hours–about 13 months. We presume that most attackers will have less than a terabyte of memory available. In such a case, their rate of progress is significantly worse–on the order of hundreds of thousands of hours.

Administrators worried about distributed cracking tools have a simple and effective defense: just grow the hash file. Space does not permit a complete discussion of table growth, but an intuitive approach is to place the old table within the larger table, and rehash each user into the larger space when they access their stored message.

Note that there are several orders of magnitude in price difference between drives and memory. This means that if adversaries attempt to match the size of the storage table with more memory, an administrator merely needs to buy more disk space. For a few thousand dollars, administrators force the attackers to spend millions to match the size of the table. This is an arms race attackers cannot easily afford.

4.2 Reliability Analysis

When shares are written to the table, there exists a chance that valid entries may be overwritten by shares for another data item. The probability of no collision *whatsoever* when inserting a total of n items, each with k shares, is computed as:

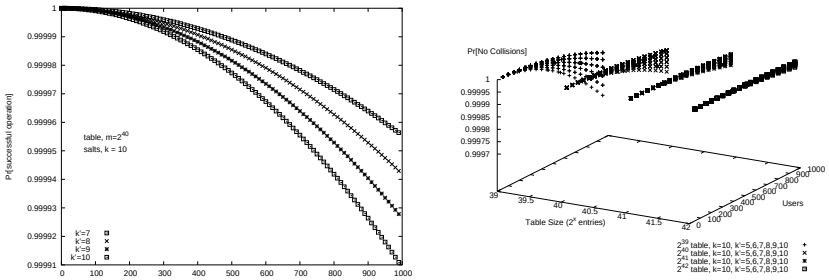
$$P_0 = \prod_{i=0}^{nk-1} \left(1 - \frac{i}{m}\right) \tag{5}$$

For practical purposes, we assume hash values are independent for a good-enough secure hash function. We can use the Equation (5) to compute for a desired probability, say 99.9999%, how many data elements (each with some k hashes) can be stored in a table with size m.

We can relax the matching requirement a bit, as long as the data has $k' \leq k$ shares in the table, the data can be retrieved. That is, for each element, we allow at most $l = k - k'$ of its shares to be overwritten by other write operations. Intuitively, we can then accommodate more data using the same table while achieving the same desired (low) probability of rejecting a valid key. The exact calculation of P_l, the probability that each data item has at least k' valid shares (i.e., no more than l shares are overwritten), is very complicated. For simplicity's sake, we can compute the lower bound of P_l. We use the following:

$$P'_l = \prod_{i=0}^{n-1} \prod_{j=0}^{k-1} \left(1 - \frac{ik' + j}{m}\right) \tag{6}$$

This can be interpreted as: when inserting the k shares for the ith data item, avoid the first k' valid shares for each of the $(i$-1)th items already in the table, and the k shares of the ith item themselves do not overwrite each other, (i.e., there is no self-collision.) It is easy to see that this calculation does not include other possible ways



(a) Collisions (b) Table Size

Fig. 3. a) The number of user data entries in a table versus the chance that no collisions occur, for a table with $m = 2^{40}$ entries, and ten salts per data item. By tolerating a few collisions, $k' < k$, higher reliability is achieved. b) The relationship between table size, item count, and successful operation. For small tables, variations in k' may be necessary to improve reliability. More tables can also be added cheaply to improve performance. Alternatively, one can restructure the table into tiers

that can lead to the condition where each item has at least k' valid shares. Therefore, P_l' is a lower bound of P_l, i.e., $P_l \geq P_l'$. It is obvious that $P_l' \geq P_0$. Therefore, we have $P_l \geq P_0$. For large m and small l, P_l' is very close to P_l. We thus use this simple estimation.

Figure 3(a) shows the benefit of allowing some collisions (up to $k = k'$ to occur). As more data is added, there's an increasing chance that one item will suffer more than $k - k'$ collisions. At some point, the risk of such failure becomes unacceptable, and larger tables or table tiers must be used. One may be tempted to lower k' even further. However, recall that if k' is too low, an adversary has a greater probability of stealing a portion of the file and obtaining all of the required shares. Specifically, if only z bytes are stolen, there is a $(\frac{z}{m})^{k'}$ chance of all an item's shares are exposed.

Conceptually, it is best to fix an error rate, estimate the maximum number of data items, and design the file size accordingly. Figure 3(b) shows the flexibility of each parameter. To obtain a fixed error rate, one can increase the size of the table. One can also adjust k and (to a lesser extent) k' to achieve the desired error rate. If one is constrained by a drive budget, and cannot find a configuration with an acceptable reliability, then table tiers provide a solution.

The VAST system also addresses the problem of malicious data corruption. If an attacker does not attempt to read the secret data, but merely tries to delete it, VAST provides two defenses. First, the attacker does not know where the secret shares are stored, so the attack must corrupt nearly a terabyte to have a chance of success. Second, if the attacker merely corrupts a fraction of the storage table, subsequent reads can detect the errors, and create new salts, thereby "healing" the table with each read. In a normal secret storage system (e.g., a password-protected file), the attacker merely has to change as little as one byte to damage the file.

4.3 Efficient Legitimate Use

To fully evaluate a security enhancement, the increased cost of an attack should be balanced against the costs imposed on legitimate use. An implementation and testing of VAST shows that it can efficiently handle many data retrieval operations per second. Each operation involves a hash computation, a seek and a read from disk. Even though retrieving information may require up to k disk reads, in practice the number of salts is small enough to make this efficient. In our tests, when all table operations require drive access, the number of operations is limited to around 250 per second per drive, using a slow (5400 rpm) IDE drive. Thus, when using low-end equipment there is an upper limit to how many records can be retrieved at a time. If one anticipates more than $250/k$ simultaneous reads, then the hash store may use faster drives, or could be distributed over a RAID system.

An important observation is that, once completely I/O bound, the performance of VAST does not decrease with larger tables. Figure 4 shows that with small tables (unacceptable from a security point of view), a good portion of the file can be cached by an operating system's I/O buffers. As a result, reads are quick, and hundreds of thousands of hash validations can be performed per second. As tables grow in size, particularly at around 2^{25} entries an above, the majority of the hash file then resides only on disk, and performance degrades. With large files, I/O comes to dominate the $find()$ operation time (which includes both I/O and memory operations for decryption and share recovery). Thus, the performance does not degrade further. Eventually, a steady rate is reached as the OS block cache becomes dominated by the drive seek time. So, one may add more terabytes to a hash store without lowering performance further. In fact, in our

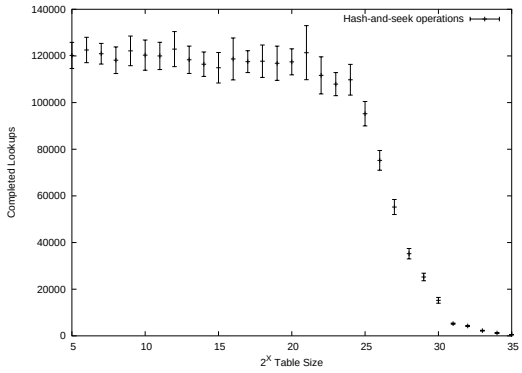


Fig. 4. Performance of a VAST system deployed on FreeBSD, retrieving random records. Due to the unpredictable location of shares on disk, and coincidental proximity of some hashes in single (8K) blocks, performance varied. Plots show the mean number of hash-seek-read operations, with standard error, compared to table size. In practice, one would use a terabyte-sized file. But the output for smaller-sized files is included to show how memory greatly speeds up performance. Significantly, even though performance degrades for larger files, it reaches a minimum of no less than 250 operations per second. Thus, one may add more terabytes to an I/O-bound VAST system, and expect no further performance degradation

testing we observed a very slight increase in performance with the addition of each new drive since each spindle provides its own independent service rate.

One might be concerned about the efficiency of reading any k' subset of k shares. That is, if the authentication phase must find the right k' of k, it could potentially take $\binom{k}{k'}$ operations. In practice, however, the first k' of the k shares will almost always provide a correct match. Even under considerable load, the system may be designed to perform with 99.9999% success. And since k and k' do not differ much and are small, around 10-15, the rare worst case scenarios will not take an unreasonable amount of work to complete.

5 Conclusion

Despite the best efforts of systems administrators, storage systems will become vulnerable, and attackers will sometimes succeed. The VAST system provides a way to store information that resists successful penetrations. In addressing this problem, this paper contributed the following points.

First, we studied the problem of protecting secret data storage against insider attacks, and formally defined it as the problem: How to store data in a table such that no fraction of the table yields useful information? Reasoning about this problem suggested the use of large storage systems to minimize the attacker's chance of success, and to increase the cost of attack.

We then proposed the VAST system as one possible solution to the secure data storage problem. Each secret has its multiple shares randomly distributed in an extremely large file. Random decoy shares and the lack of usable identification information prevent selective copying or analysis of the file. No single part of the file yields useful information in isolation from the rest. The file's size and structure therefore present an enormous additional hurdle to attackers attempting to transfer, steal or analyze the data.

Finally, we implemented the VAST system, and demonstrated that it performs reasonably well for normal use. Experiments show that breaking VAST requires an enormous amount of time and resources. Under our security model, VAST greatly improves the security of data storage as well, since attacks are likely to trigger an alert and response. Unlike previous work, e.g., [FDP91, FDR92, DFF+88], VAST requires only a single host, and presumes an attacker may access the protected file.

Using large files to safely store data is a counter-intuitive approach to security. VAST demonstrates how algorithms that maximize the number of I/O operations can be used to improve security, similar to traditional fragmentation-and-scattering schemes. With affordable tera-scale storage devices, we believe solutions to the table storage problem now have many practical applications.

References

[AKL93] Lars Arge, Mikael Knudseni, and Kirsten Larsent. A general lower bound on the complexity of comparison-based algorithm. In *Proceedings of the 3d Workshop on Algorithms and Data Structures*, volume 709, pages 83–94, 1993.

[AV87] A. Aggarwal and J.S. Vitter. The i/o complexity of sorting and related problems. In *Proc. 14th ICALP*, 1987.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley-Longman, 2003.

[CFIJ99] Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In *Proc. of STACS 99*, 1999.

[Che90] B. Cheswick. The design of a secure internet gateway. In *Proc. of Usenix Summer Conference*, 1990.

[CSGV81] R.M. Capocelli, A. De Santis, L. Gargano, and U. Vaccaro. On the size of shares for secret sharing schemes. *Lecture Notes in Computer Science*, 576, 1981.

[DFF+88] Y. Deswarte, J.C. Fabre, J.M. Fray, D. Powell, and P.G. Ranea. Saturne: a distributed computing system which tolerates faults and intrusions. In *Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, pages 329–338, September 1988.

[FDP91] J.-M. Fray, Y. Deswarte, and D. Powell. Intrusion tolerance using fine-grain fragmentation-scattering. In *Proc. IEEE Symp. on Security and Privacy*, pages 194–201, 1991.

[FDR92] Jean-Charles Fabre, Yves Deswarte, and Brian Randall. Designing secure and reliable applications using fragmentation-redundancy-scattering: an object-oriented approach. In *PDCS 2*, 1992.

[FM03] Holly Frost and Aaron Martz. The storage performance dilemma. `http://www.texmemsys.com/files/f000160.pdf`, 2003.

[HP03] John L. Hennessy and David A. Patterson. *Computer Organization and Design*. Morgan Kaufman Publishers, 2003.

[KP00] Stefan Katzenbeisser and Fabien A. P. Petitcolas, editors. *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House Books, 2000.

[LAV01] Subramanian Lakshmanan, Mustaque Ahamad, and H. Venkateswaran. A secure and highly available distributed store for meeting diverse data storage needs. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[Mau92] Ueli M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. 1992.

[MR98] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213, 1998.

[Pat94] N.P. Patt. The I/O subsystem: A candidate for improvement. *IEEE Computer: Special Issue*, 24, 1994.

[PM99] Niels Provos and David Mazieres. A future-adaptable password scheme. `http://www.openbsd.org/papers/bcrypt-paper.ps`, 1999.

[Rab89] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36, April 1989.

[Sha70] A. Shamir. How to share a secret. *Comm. of ACM*, 13(7):422–426, 1970.

[WBS+00] Jay Wylie, Michael Bigrigg, John Strunk, Gregory Ganger, Han Kiliccote, and Pradeep KhoslaComputer. Survivable information storage systems. In *IEEE Computer*, volume 33, pages 61–68, August 2000.