

# Project TWOVAULT – Secure and Selectively Deniable Data Storage

Markku-Juhani O. Saarinen *Member, IEEE*

**Abstract**—We introduce TWOVAULT, a cryptographic data storage mechanism with novel features that has been designed primarily for securing removable memory devices and Solid State Disks. The design is organization-centric and therefore suitable for large corporations and governmental organizations that wish to control the access and flow of sensitive information.

In TWOVAULT, data confidentiality and integrity protection are achieved with *Envaulting*, a standards-based cryptographic mechanism for securing stored data with keyless access control. Envaulted removable devices can be remotely disabled if the device is stolen or lost. Furthermore TWOVAULT makes attempts to access envaulted data traceable by its owners.

In Envaulting, cryptographically diffused data is split into two parts of unequal size. The larger portion is stored in a container (e.g. a USB Flash Drive), while the smaller portion, together with keying information, is stored in a separate fragment database, the FragmentVault Server. After establishing a secure connection with FragmentVault, the user sees files stored in the data container as part of the regular file system, making the system transparent and easy to use.

Encrypted blocks in the data containers are indexed by a FragmentVault server using a mechanism that gives it a high degree of deniability; TWOVAULT is a deniable steganographic filesystem. The total amount of information stored on a device remains unknown to an adversary even if portions of it has to be revealed due to legal or extralegal pressure.

The Content-Addressed Storage (CAS) mechanism of TWOVAULT often eliminates block duplicate writes on the container (removable storage device), speeding up common operations, increasing capacity and device lifetime. The indexing system also allows “undo” operations; recovery of earlier snapshot views the filesystem.

A fully operational prototype of TWOVAULT has been implemented for the Linux operating system.

## I. INTRODUCTION

We shall first introduce the concepts and motivational observations behind TWOVAULT.

One of the motivating observations in the design of TWOVAULT is the widely predicted and already occurring transition from spinning magnetic disks to solid state storage. Filesystems have traditionally been designed to optimize the storage of files onto continuous blocks on disks to enable fast retrieval. One of the main differentiating characteristics of Flash and Solid State Drive (hereafter simply SSD) storage solutions is near-zero latency and seek time. Usage of defragmentation tools on SSDs makes little sense as blocks on these drives can be accessed in random order. We note that NAND flash memory does not behave exactly like RAM; the underlying technology often necessitates that information is

erased one allocation block at a time, thus making byte-level random access slow.

Another major difference between traditional disk systems and SSD systems is that the latter may only offer a limited number number of erase/write cycles. Some vendors only guarantee 10000 writes before errors may occur. We address this problem by randomizing the use of a disk across the entire physical drive space. TWOVAULT has near-optimal wear-leveling characteristics, thus increasing drive lifetime.

### A. Envaulting

The essential feature of the Envaulting concept is that some *a priori* knowledge of complete plaintext contents is required to decrypt a block of data – even if the secret key is known. The entropy of a message is split into two halves of unequal size; the larger portion can be stored or transmitted on high-capacity medium, and the much smaller portion on low-bandwidth secure channel. Both portions are required to reconstruct the original message and can be used to verify its integrity.

In the TWOVAULT instantiation of Envaulting the high-capacity medium is a SSD and the low-bandwidth secure channel is an online connection to a “FragmentVault” Server. Other instantiations might use a fast internet connection or shared disk space as the insecure channel and a quantum link or a smart card as the secure low-bandwidth channel.

There are many ways to perform Envaulting, but in the present work we use a cryptographic hash function and a tweakable block cipher mode of operation as the Envaulting construction (see Figure 1). The Envaulting operation is discussed in detail in Section II-B.

The theoretical background of Envaulting can be traced to Claude Shannon’s 1948 work on information theory; in some ways the low-capacity channel acts as an Error Correcting Code that allows decoding of the data in the high-capacity medium (see Section 12 in [1]). However, due to use of a cryptographic diffusion layer, no partial information about plaintext is leaked unless both portions, together with a third element, a symmetric cryptographic key, are fully available.

### B. Resistance to Key Disclosure

Most traditional encryption systems encrypt the entire protected volume with a single key that is derived from a passphrase or password in some way. Dictionary attacks against the passphrase cannot be mounted against envaulted volumes.

Recent research has found many (if not most) disk encryption systems vulnerable to so-called *cold-boot* attacks

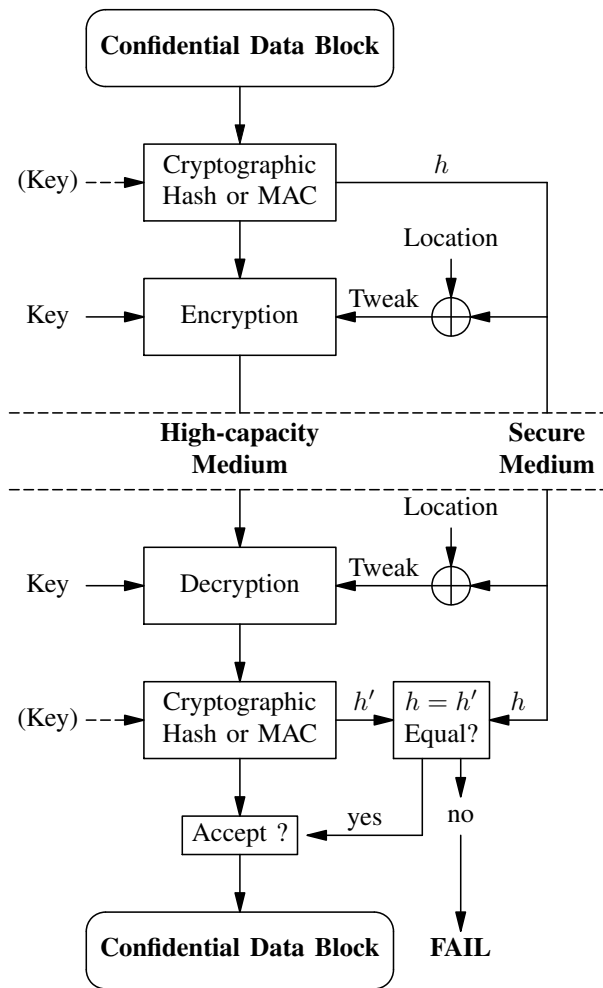


Fig. 1. Envaulting and devaulting using a hash function. Use of a hash function as a tweak guarantees that the ciphertext is fully diffused. Decryption of a block is not possible without the hash value  $h$ , even if the secret encryption key is known to the adversary.

[2]. Cold-boot attacks are based on the observation that the contents of system memory degrade slower than previously thought, allowing attackers to remove physical memory and copy passwords and encryption keys from system memory.

Even a full snapshot disclosure of encryption keys and Fragment Vault contents does not allow decryption of modifications performed after the snapshot has been made. Envaulted storage therefore has *forward secrecy* properties, as applicable to static storage.

### C. Content-Addressed Storage and Version Control

TWOVAULT implements a CAS (Content-Addressed Storage) scheme. If two bulk data blocks (of size that is equal to allocation unit, typically 4096 bytes) contain exactly the same data, they are stored on the medium only once. This is achieved using a collision-resistant hash function (see Chapter 9 of [3]).

We utilize a simple queuing mechanism for block allocation which guarantees that blocks are not immediately reused. Therefore previous snapshot views of the entire filesystem can be recovered by simply altering the virtual block indexing system.

### D. Selective Deniability

TWOVAULT introduces information hiding with an implementation of a highly efficient steganographic storage system based on ideas of Anderson, Shamir, Needham and others [4], [5]. This feature can be used to hide not only the plaintext contents, but the very existence and amount of information stored on a disk. Multiple hidden filesystem containers can exist on a single removable media. Even if an adversary obtains access to one such container, it does not reveal any information about *additional* information stored on the device. Proving that the storage device holder has not provided access to all information contained in the storage device can be shown to be impossible.

### E. Organizational Remote Control

With TWOVAULT, each read and write operation must be individually authenticated. In our implementation, an authenticated SSL/TLS connection to a trusted server is required to store and retrieve deniability, confidentiality and integrity information. Each individual block read and write operation can be logged on a trusted secure server. This allows access to memory devices to be centrally controlled, and dictionary attacks can be proactively prevented. A lost, stolen, or decommissioned memory device can be simply remotely disabled. Any attempt to access it, with or without the authentication keys, can be detected and even traced.

## II. IMPLEMENTATION

TWOVAULT utilizes a client-server model where bulk data is stored on a SSD memory device by the TWOVAULT client, and indexing, authentication, and integrity validation information on a remote server, FragmentVault. The communication link between a TWOVAULT client and FragmentVault is secured using the SSLv3 or TLS protocol [6].

In the prototype Linux version of TWOVAULT, the client side is implemented as a user-level process that listens to incoming Network Block Device (NBD) connections on a local socket. This allows installation without modifications to current stock Linux Kernel. This is illustrated in Figure 2.

### A. Binary Protocol

We currently use a very light-weight binary protocol over SSL/TLS. The OpenSSL library [7] is used to implement AES, RIPEMD-160 and TLS/SSL. The “https” port 443 is used by default as typical firewall configurations allow outgoing traffic through this port. FragmentVault is authenticated using a trusted site X.509 certificate. The TWOVAULT client may use any authentication mechanism allowed by the local OpenSSL installation, including password authentication, (smart card) certificate authentication, and various two-factor authentication schemes.

The binary protocol is not used to transmit data blocks stored on the bulk memory device, but rather their message digests (hashes) and indexing information. Even if the communication link is somehow breached, there is no actual confidential data leakage. Figure 3 illustrates the messages in the current binary protocol.

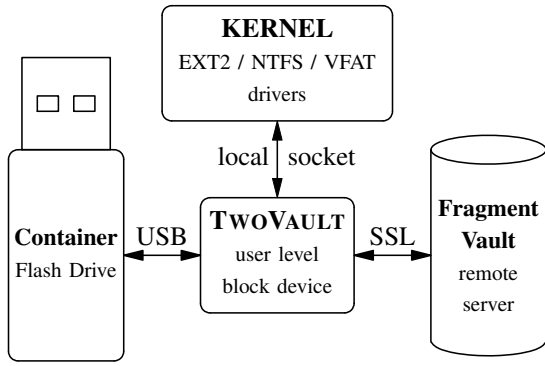


Fig. 2. Kernel file systems communicate with TWOVAULT via a local NBD socket. TWOVAULT encrypts and stores data on a bulk data device (here a USB flash drive) while communicating with FragmentVault.

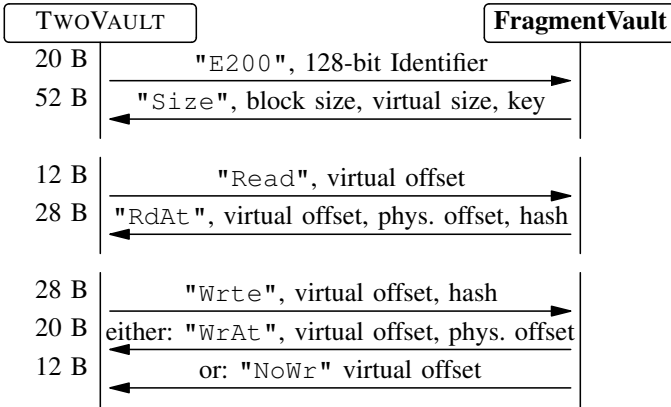


Fig. 3. Illustration of the current version of the binary protocol, which is encapsulated inside an authenticated TLS connection. All sizes and offsets are 64-bits in size, hashes are 128 bits, key is 256 bits and big-endian (network) byte order is used for numbers.

### B. Confidentiality and Integrity Mechanism

TWOVAULT confidentiality protection (encryption) is based on the standardized Advanced Encryption Standard (AES) algorithm in the XTS mode of operation, as specified in FIPS 197 and IEEE P1619 standards [8], [9]. The XTS mode tweak is specified as the XOR of the cryptographic hash of the plaintext block and physical block index.

RIPE-MD160 [10] hash function is used due to reported problems with security of MD5 and SHA hashes. After NIST completes its SHA-3 selection process, we plan to migrate to the chosen SHA-3 algorithm [11].

We use the following symbols and variables in the description of the Envaulting encryption mode used in TWOVAULT:

- $\oplus$  Bitwise XOR operation.
- $\otimes$  Multiplication in the finite field  $\text{GF}(2^{128})$  defined by the polynomial  $x^{128} + x^7 + x^2 + x + 1$ .
- $H$  Collision-resistant hash function RIPEMD-160.
- $E_k$  AES Encryption with key  $k$ .
- $D_k$  AES Decryption with key  $k$ .
- $K$  Secret key. Split into two halves:  $K = K_1 \mid K_2$ .
- $i$  Physical block index.

$j$  Index within the block. For 4096-byte blocks,  $0 \leq j \leq 255$ .

$\alpha$  Generator in  $\text{GF}(2^{128})$  i.e.  $x$ .

$P_j$  128-bit plaintext block.

$C_j$  128-bit ciphertext block.

Encryption and decryption is always performed on full filesystem allocation unit blocks, typically 4096 bytes in size. To encrypt a block, the following steps are required.

1. Compute the cryptographic hash of the message.  
 $h = H(P_0 \mid P_1 \mid \dots \mid P_{n-1})$
2. Find its physical index on device as discussed in Section II-C. Encryption and write operations can be avoided if the data contents are already stored on the device.  
 $i = \text{find\_write\_loc}(h)$ ;
3. Compute the tweak mask  $T$ .  
 $T = E_{K_2}(h \oplus i)$ .
- 4e. Encrypt each 128-bit block  $i = 0, 1, \dots, n-1$ .  
 $C_i = E_{K_1}(P_i \oplus (T \otimes \alpha^i)) \oplus (T \otimes \alpha^i)$

The finite field multiplication is very fast; only one shift and conditional XOR by a 128-bit mask value is required for computation of  $T \otimes \alpha^i$  if  $T \otimes \alpha^{i-1}$  is known. XTS encryption is illustrated in Figure 4.

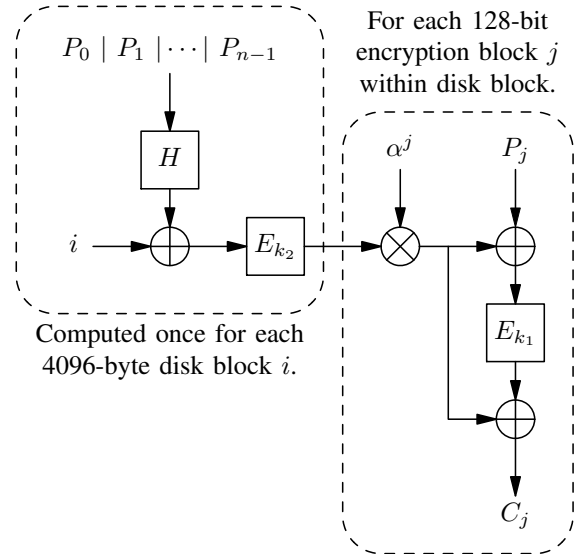


Fig. 4. The IEEE 1619 XTS mode is “tweaked” with the hash of the plaintext.

For decryption operation to work, the  $h$  value must be available – it is obtained from the FragmentVault and also used to verify the integrity of the plaintext. Encryption and decryption processes are the same, except for the last step 4:

- 4d. Decrypt each 128-bit block  $i = 0, 1, \dots, n-1$ .

$$P_i = D_{K_1}(C_i \oplus (T \otimes \alpha^i)) \oplus (T \otimes \alpha^i)$$

Integrity is verified by computing the hash of the resulting decrypted plaintext and comparing that to the  $h$  value used in decryption.

Using a standardized cipher and a mode of operation that has been specifically standardized for protection of data on storage devices, rather than a proprietary system, allows data to be recovered even in distant future when the software tools and platform that were used to store data are no longer easily available. The P1619 standard has a XML-based mechanism

for storing secret keys and tweaks. TWOVAULT can be made to support this mechanism for permanent storage.

Another feature of the XTS mode of operation when compared to more classical modes of operation is its inherent parallelism. Encryption using the CBC mode requires the ciphertext of the previous block in order to encrypt the next one, thus breaking parallelism. We envision standardized hardware components that use AES in XTS mode of operation. For discussion about the security of XTS mode of operation, see section IV-A.

### C. Indexing of Blocks by FragmentVault

Data blocks are indexed by a map based on the hash of the data content of the block. Each unique block is only stored once; if a block of data with given contents already exists on disk, it is unnecessary to write another copy of it. It is sufficient to simply update tables that map virtual block indexes to physical block indexes.

Indexing is based on RIPEMD-160 [10] truncated to 128 bits due to storage considerations. By birthday paradox, collisions are not likely to occur before  $2^{64} \approx 1.845 \cdot 10^{19}$  blocks have been written on disk. This, together with 64-bit sector addressing, allows the current solution to be comfortably expanded up to exabyte range. [3]

The mapping of virtual block positions to physical block positions is performed via two tables, `hmap` and `pmap`. The array `hmap` is a table of hashes (type `md_t`) corresponding to each virtual block. The `pmap` table maps physical indexes and maintains information that is used for hash searches and allocation of physical disk blocks. Figure 5 illustrates how these data structures are used.

The `pmap` structure is internally defined as:

```
typedef struct {
    md_t      md; // hash of a block
    uint64_t  pcnt; // block usage count
    uint64_t  scnt; // for hash search
    uint64_t  ploc; // location on disk
} pmap_t;
```

We will now briefly describe the hash table algorithms used. All of these algorithms have  $O(1)$  speed, therefore being faster than e.g.  $O(\log n)$  binary search algorithms. No sorting is required.

1) *Block allocation / deallocation*: A ring queue `rngq` of  $n$  elements is used for physical block allocation and deallocation. The ring queue is initialized with the indices of all blocks contained in the particular volume; they can be in random order. Also,  $n$  doesn't have to correspond to physical disk size in blocks. The deniability (information hiding) feature of TWOVAULT (see [4]) relies on this property. Two index variables, `rqtop` and `rqend` maintain the top and bottom of the queue, in wrap-around fashion. New blocks are obtained from the top of the queue with

```
new_block = rngq[rqtop++];
if (rqtop >= n) // wraparound
    rqtop = 0;
```

Unused ones can be freed by inserting them at the queue end:

```
rngq[rqend++] = old_block;
```

```
if (rqend >= n) // wraparound
    rqend = 0;
```

We chose a queue mechanism rather than a stack mechanism to facilitate the “undo” feature; recently overwritten blocks are not immediately reused. Undo is achieved by simply reversing changes caused by write operations in the FragmentVault by using a write log. This also acts as a wear-leveling mechanism.

2) *Finding a block of data by hash*: Cryptographic hashes are uniformly distributed regardless of input. We linearly scale the hash result range onto the size of `pmap` to get the initial probe location as follows: The first eight bytes of the hash are converted into a 64-bit integer  $x$ , which is divided with a constant  $d$ . The `pmap` structure therefore has  $\lceil 2^{64}/d \rceil$  entries. In the current implementation the constant is chosen as  $d = \lfloor 2^{64}/(1.2n) \rfloor$ , where  $n$  is the size of the physical disk.

The initial index  $i = \lfloor x/d \rfloor$  is called the “slot” of the hash  $h$ . When the physical disk fills up, the `pmap` table also fills up and two or more hashes try to use up the same slot  $i$ . The count of hashes in each slot is maintained in `pmap[i].scnt`.

```
uint64_t slot(h) {
    return *((uint64_t *) h) / d;
}
```

Pseudocode for the algorithm `find_by_hash()` for finding the entry in `pmap` that corresponds to hash  $h$  is given by.

```
s = slot(h) // our slot
c = pmap[s].scnt; // number of entries
i = s; // index
while (c > 0) {
    if (h == pmap[i].md)
        return i;
    // same slot? decrease count
    if (slot(pmap[i].md) == s)
        c--;
    if (++i >= pmapsiz) // wrap
        i = 0;
}
return NOT_FOUND
```

3) *Reading a Block*: When TWOVAULT receives a request to read from virtual location  $i$ , it simply translates it into a physical location  $p$  using the FragmentVault connection and performs a actual read operation. In FragmentVault:

```
p = find_by_hash(hmap[i]);
return pmap[p].ploc;
```

Note that if nothing has been written to block  $i$  yet, its `htab` entry is zero. In such a case FragmentVault simply returns “Not Available” and TWOVAULT writes a block of zeros to the bulk store.

4) *Writing a Block*: Write operation is slightly more complicated than the read operation. To write a block of data given by `*block` at virtual location  $i$  we do the following

```
h = hash(block); // hash it
if (hmap[i] != NOT_USED) {
    if (h == hmap[i])
        return; // NOP!
    j = find_by_hash(hmap[i]);
    // do we need to free this one?
    if (--pmap[j].pcnt == 0) {
        pmap[slot(hmap[j])].scnt--;
        rngq[rqend++] = omap[j].ploc;
```

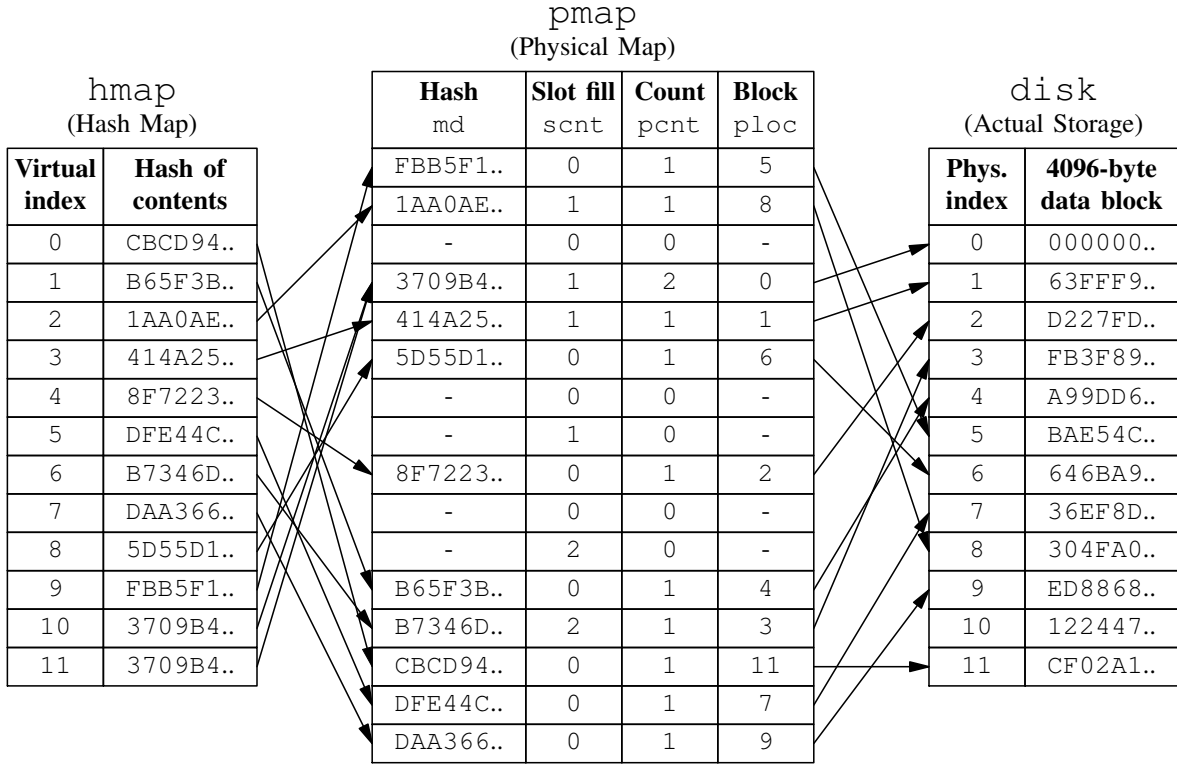


Fig. 5. An example of mapping a virtual block index into physical index via hmap and pmap inside the FragmentVault, also illustrating the hash table mechanism used. Blocks with the same content (10 and 11) are mapped to the same physical sector 0 in the picture.

```

if (rqend >= n) // wrap
    rqend = 0;
}
}
hmap[i] = h; // set it
j = find_by_hash(h);
if (j != NOT_FOUND) {
    pmap[j].pcnt++; // counter
    return; // done!
}
s = slot(h); // our slot
pmap[s].scnt++; // new
j = s;
while (pmap[j].pcnt > 0) {
    if (++j >= pmapsiz) // wrap
        j = 0;
}
pmap[j].md = h; // set the hash
pmap[j].pcnt = 1; // usage=1
// do NOT touch scnt!
pmap[j].rngq[rqtop++];
if (rqtop >= n) // wrap
    rqtop = 0;

```

### III. PERFORMANCE ANALYSIS

In FragmentVault, the read operation is a simple table lookup and causes minimal effect on performance. The write operation involves a hash function computation and an  $O(1)$  insertion algorithm. Since the algorithm eliminates a large proportion of media I/O required in the typical usage, there are significant speedups in write operations, especially with storage devices with relatively slow write speeds. The communication bandwidth required is minimal (less than 1 %) when

compared to device I/O or protocols such as the Network File System [12].

To estimate the storage requirements in FragmentVault, we give the ratio of the additional data structures to the actual stored data in our prototype implementation:

hmap	16 Bytes
pmap.md	16 Bytes
pmap.pcnt	8 Bytes
pmap.scnt	8 Bytes
pmap.ploc	8 Bytes
pmap total	40 Bytes
rngq	8 Bytes

$$\text{Fraction} = \frac{16 + 1.2 \times 40 + 8}{4096} \approx 1.758\%$$

However, probability that search count should exceed  $c$  when pmap expansion factor 1.2 is used is given by  $1.2^{-c}$ . Hence only one byte is sufficient for scnt storage on essentially any future system as  $1.2^{-256} \approx 2^{-67}$ . By using 16k blocks and 32-bit addressing (max capacity 64 TB), and 72-bit hashes, the fraction can be reduced to 0.207%; 2.1 MB of tables in FragmentVault per each GigaByte stored on the removable device.

### IV. SECURITY ANALYSIS

Finally, we will present short security arguments for TWOVAULT. These can be extended into security proofs in many realistic security models.

### A. Confidentiality

In [13] Rogaway develops a tight security proof for XEX/XTS showing that if the underlying AES block cipher is secure against a powerful chosen plaintext/ciphertext adversary, so is the resulting XEX/XTS mode of operation. We note that AES has been approved by NSA to carry data up to Top Secret level in Governmental Use. We apply the XTS mode using the hash of the plaintext message as part of the tweak. This extra layer of protection increases the diffusion from the input bits across the entire allocation unit “block”.

### B. Integrity

Our solution provides strong integrity protection for stored data using the RIPEMD-160 cryptographic hash. Most of the traditional disk encryption systems do not provide any integrity protection; random manipulation of ciphertext will not be detected, but simply be visible as corrupted “nonsense” when decrypted.

### C. Access Control

In traditional encrypted filesystems the access to data is dependent only on a single user-specified key. In our solution on-line authentication is required to access each sector of envaulted data. Therefore off-line dictionary attacks are simply not feasible. Advanced access control mechanisms such as security tokens or smart cards can be used to authenticate the SSL/TLS connection, and all access to the device is traceable by the FragmentVault’s audit trail.

### D. Selective Deniability

The TWOVAULT system provides selective deniability, even in forensic analysis. It can be seen that it is possible to hide data on the drive in a way that prevents the very existence of certain hidden volumes to be revealed even if FragmentVault authentication is successful. However, the data is not deniable to the organization that controls FragmentVault and owns the device. This was one of the original design criteria of TWOVAULT.

## V. ACKNOWLEDGMENT

The author is deeply thankful to colleagues at Envault Corporation for the original inspiration, encouragement and support for “Project TWOVAULT”.

## REFERENCES

- [1] C. E. Shannon, “A mathematical theory of communication,” *The Bell System Technical Journal*, vol. 27, pp. 379–423,623–656, Oct. 1948.
- [2] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest we remember: Cold boot attacks on encryption keys,” in *Proc. 17th USENIX Security Symposium*. USENIX Association, 2008, pp. 45–60.
- [3] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.
- [4] R. Anderson, R. Needham, and A. Shamir, “The steganographic file system,” in *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings*, ser. Lecture Notes in Computer Science, D. Aucsmith, Ed., vol. 1525. Springer, 1998, pp. 73 – 82.

- [5] A. D. McDonald and M. G. Kuhn, “Stegfs: A steganographic file system for linux,” in *Information Hiding, Third International Workshop, IH’99, Dresden, Germany, September 29 - October 1, 1999, Proceedings*, ser. Lecture Notes in Computer Science, A. Pfitzmann, Ed., vol. 1768. Springer, 2000, pp. 463–477.
- [6] T. Dierks and E. Rescorla, “The transport layer security (TLS) protocol version 1.2,” 2008. [Online]. Available: <http://tools.ietf.org/rfc/rfc5246.txt>
- [7] OpenSSL, “OpenSSL: The open source toolkit for SSL/TLS.” [Online]. Available: <http://www.openssl.org>
- [8] NIST, *FIPS 197, Specification for the Advanced Encryption Standard (AES)*, Nov. 2001. [Online]. Available: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- [9] IEEE, *Std 1619-2007: Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*. IEEE, Apr. 2008.
- [10] H. Dobbertin, A. Bosselaers, and B. Preneel, “RIPEMD-160: A strengthened version of ripemd,” in *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1039. Springer, 1996, pp. 71 – 82.
- [11] NIST, “Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family,” *Federal Register*, vol. 72, no. 212, pp. 62 212–62 220, Nov. 2007.
- [12] S. e. a. S, “Network file system (NFS) version 4 protocol,” 2003. [Online]. Available: <http://tools.ietf.org/rfc/rfc3530.txt>
- [13] P. Rogaway, “Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac,” in *Advances in Cryptology – Asiacrpt 2004*, ser. Lecture Notes in Computer Science, vol. 3329. Springer, 2004, pp. 16–31.



**Markku-Juhani O. Saarinen**, CISSP-ISSAP, received his M.Sc. degree in Mathematics and Computer Science from University of Jyväskylä in Finland and is currently pursuing a Ph.D. degree in Information Security with Royal Holloway, University of London. He works as a Cryptography Specialist with Envault Corporation, Finland.

Mr. Saarinen was involved in the development of SSH2 and IPsec secure communication protocols in late 1990’s. Since then, he has worked in varied research and development roles in the Information Security field with SSH COMSEC, Nokia Research Center, Helsinki University of Technology (with funding from Finnish Defence Forces), NIXU Middle East (Dubai / Riyadh), and Envault Corp. Besides his native country of Finland, his security work and research has taken him to work and live in the United Kingdom, Saudi Arabia, United Arab Emirates, Kuwait, and Spain.