I J C T A, 9(37) 2016, pp. 63-74
© International Science Press

Clean Slate: A Secure Virtual File System with Device Authentication for Mobile Devices

Partha Sarathi Chakraborty*, **Kaustubh**** **and Raghavendra Pratap Singh*****

Abstract: This paper addresses the need of data access control and privacy on an occasion of device theft/loss or any other scenario where the attacker gains physical access to the device. Theft/Loss of mobile storage devices is a pressing issue as of present as the number of mobile devices' users increase. Furthermore user's ignorance towards security only makes matters worse. Clean slate is an attempt at dealing with the after effects of such mishaps in a way that's easy and user friendly. Clean slate is a FUSE (File System in User Space) based cryptographic virtual file system. It transparently encrypts the data locally, but stores the secret keys remotely, away from the device itself, managed by a key service. We also include the concept of trusted device pairing, via the usage of digital certificates so that the device itself can detect if it is lostorstolen and deny access to remotely stored decryption keys and consequently the data. If all else fails we need to know if any data has been compromised and for that we propose a logging service. This way we are able to take better informed decisions. If it is deployed correctly, this can be a very effective solution, but with some impact on I/O performance, which we have attempted to address in this paper.

Keywords: Mobile, Secure, Theft, Device Pairing.

1. INTRODUCTION

Device theft/loss is a common and persisting problem as their usage grows more and more.56% of US citizens misplace their cell phone or laptop each month.113 cell phones are lost or stolen every minute in the U.S. About 12000 laptops are stolen each week in US airports alone. The fact that most of users don't understand security only makes the situation worse. A survey by Consumer Reports suggests that 36% of users lock their mobile screens with a 4 digit PIN. Only 7% use security features such as encryption and 34% of users don't take any security measures at all. According to the paper "Why Johnny can't encrypt" the reason why users avoid using encryption is simply because they don't understand it and they can't use it as it is not user friendly. Also available solutions for mobile devices seem to be ineffective in a situation where the attacker has physical access to the device as they can easily be uninstalled/disabled/bypassed.

Furthermore present security solutions make use of passwords in way or another. But the users usually tend to use passwords that are easy to remember and often end up using easily guessable passwords. They also don't change them very often. A report by Splash Data compiled from a list of leaked compromised passwords suggests that the most commonly used passwords are "123456", "password", "12345678", "qwerty", "football", "batman" and the likes.

Here, let's consider a few scenarios where Clean Slate can be useful. Let's assume Alice is an employee at Mega Corp. She has a USB drive with critical financial details on it. The IT department had set it up with Clean Slate. It gets stolen by a rival company while she was at a party. Somewhere Bob plugs it into his laptop. The device refuses to pair with his laptop as he does not have a valid certificate installed on his laptop. The logging service logs the IP address of the host machine and sends an email/SMS alert to Alice.

* Assistant Professor, Department of Computer Science & Engineering, SRM University, NCR Campus, Ghaziabad, India

** B. Tech. CSE, Department of Computer Science & Engineering, SRM University, NCR Campus, Ghaziabad, India. *Email: kaustubh0163@gmail.com*

*** B. Tech. CSE, Department of Computer Science & Engineering, SRM University, NCR Campus, Ghaziabad

Alice realizes that she has lost the device and informs the IT department as well as law enforcement that can now use the IP address to locate Bob.

Let's consider another situation like above, only this time Alice realizes the device is lost/stolen but Bob is able to obtain Alice's digital certificate and gets past host authentication. But since Alice has already informed the IT department and the IT department has flagged the device as stolen, Bob is refused access to the encryption keys. Again the logged IP address can be used to locate Bob.

In the worst case scenario when Alice isn't aware and Bob has circumvented both pairing and data access control, the logging system will at least inform the IT department when and what files were accessed. Considering the issues above we have attempted at presenting a solution that tackles these issues in a user friendly way by making the encryption and decryption process transparent to the user and minimizing the user interaction required for it to work. We encrypt data locally, but store the encryption keys remotely, thus controlling data access in occasion of theft or loss. We also propose device pairing, so that even when the user is not aware of the theft/loss the device can detect it when it is used in an untrusted environment and detect that it is stolen/lost based on which further actions can be taken.

2. RELATED WORK

Secure file systems have been around for some time now. Microsoft's EFS, the FUSE based EncFS, ecrypt file system for Linux platforms. "TransCrypt: Design of a Secure and Transparent Encrypting File System" by Satyam Sharma presents a file system with operations transparent to the user. Here we present the analysis of major cryptographic filesystems that are being used worldwide. We are mostly concerned with their cryptographic features, ease of usage, and how well they work if the attacker gains physical access to the data.

CryptFs: It is a virtual filesystem that is stacked on top of another virtual or real filesystem. It is implemented as a Virtual Node interface.

Key management: The keys are derived from a user provided passphrase that must have a minimum length of 16 characters. This is a pitfall, as a 16 character password, while strong is hard to remember and a hassle to type as well. The user hence might note it down somewhere in a text file or a paper etc. The password itself though is hashed using MD5, but if it's a compromised password of some other user, it may be cracked using rainbow tables. This is the only line of defense that prevents unauthorized decryption of data and it seems it is highly vulnerable due to user's mistakes. Also it is a hassle to enter the passphrase to decrypt the data and remember the passphrase. This makes it unfriendly.

EncFS: This is a virtual filesystem in userspace implemented using FUSE and openssl. This is the closest filesystem to ours, but still there are some shortcomings. It provides multiple cryptographic algorithms to choose from, like AES, blowfish etc. But like most others, it too requires a password to allow for decryption.

Ecryptfs: This is more or less similar to the other cryptographic filesystems we have discussed yet, it is run in kernel space as a virtual filesystem and the passphrase used to secure the keys is salted and iteratively hashed 65,537 times by default. It makes use of the Linux kernel key ring service, the kernel cryptographic API, the Linux Pluggable Authentication Modules (PAM) framework, OpenSSL/GPGME, the Trusted Platform Module (TPM), and the GnuPGkey ring in order to make the process of key and authentication token management seem seamless to the user.

TPM

The Trusted Platform Module is a hardware chip that restricts access to encryption keys based on a secure state of the system it is used in. The definition of this secure state is stored in the Platform Configuration Register at boot time. It is designed and maintained by the Trust Computing Group (TCG). Since it

recognizes a secure state, hence it becomes difficult for an attacker to access keys and its data even when he has physical access, as TPM won't give permission to an unknown application run by the attacker. But it's vulnerable to following attacks:

TPM Reset Attack

In this attack, it is assumed that the attacker is capable of monitoring the trusted state measurements sent to PCR by the BIOS and thus zeroes out the PCR thus modifying the definition of the present trusted state to be null. The system is accessed remotely as the keyboard and mouse are reset when this zeroing out happens. Hence, they are unusable. The exploit is performed locally on the hardware by grounding the LPC bus' reset line.

Physical Tampering

Christopher Tarnovsky at 2010 Black Hat conference demonstrated how the electrical signals within the chip can be manipulated and encryption can be bypassed by removing the chip's protective plastic casing and the RF-mesh. This attack is extremely sophisticated and requires a strong technical knowhow.

Conclusion: While all these technologies prevent unauthorized access to data in the event that an attacker has gained remote access to the device or somehow has managed to transfer the encrypted data, they do not provide security in an occasion where the attacker has stolen the device itself and has unlimited access to it for an indefinite amount of time. This may be more than enough to break a weak password even if it is strongly hashed. Here we again like to point out that none of these technologies take into account an average person's knowledge and motivation towards stronger security. Weak passwords currently plague IT security, and we have tried to address that.

3. DESIGN ISSUES

Design Goals

1. Confidentiality and data protection
2. Ease of use
3. Requirement of no or minimum user intervention
4. Elimination of passwords
5. Secure against user's negligence and lack of knowledge.

Not our design goals

1. Data Authentication: We choose to ignore data authentication as we don't care if the attacker modifies data or compromises its authenticity as long as he is not able to compromise security. Adding authentication mechanisms like MACs etc impact the file system's performance and aren't really necessary in this scenario.
2. Device Retrieval: We are not concerned with whether the device is returned to owner. Only that the data is not compromised
3. Data Retrieval: We are again, not concerned with retrieval of data that was stored on the device.

4. ARCHITECTURE

Clean slate is a virtual filesystem, based in the FUSE API that runs in userspace. That means, it is mounted over an on device pre-existing filesystem like EXTx, FAT etc. Any requests to the underlying filesystem are first intercepted by Clean Slate, reinterpreted, and then passed on to the underlying filesystem.

Creating a new filesystem from scratch is a slow and daunting task. Here, we are not proposing a new filesystem layout, or a new way to write and retrieve data. Clean slate is more like an add-on/extension to present/future filesystems. This is why we present this as a virtual filesystem instead of a real filesystem. Secondly we are developing it to be used in the userspace, and not kernel space. This is because developing a kernel module is difficult and slow. A crashing kernel module results in a kernel panic and warrants for a reboot. Also, a kernel module runs with administrative privileges. This opens door for some serious security concerns, which defeats the purpose of this paper. A filesystem in userspace does not require to be mounted by a privileged user. Thus even in a multiuser environment, it's usage becomes pretty safe, secure and hassle free, since the system administrator does not have to intervene at every interaction, and a user's privileges does not have to be elevated. This is especially useful in corporate/ industrial setups where a multiuser setup is extremely common.

FUSE API: FUSE is an open source library to create virtual filesystems in userspace. The library is available for many platforms (although more dominantly in Linux) and many languages like C, python and Java.

The underlying architecture of the library is divided into two parts, the kernel module (fuse.ko) and the userspacefilesystem itself. The kernel module is loaded at boot time/any time before the filesystem starts and acts as a bridge between the filesystem and the actual kernel filesystem interface. Clean slate connects to the kernel module through a socketpair, and establishes a server-client relationship, the server being the kernel module, and Clean Slate the client.

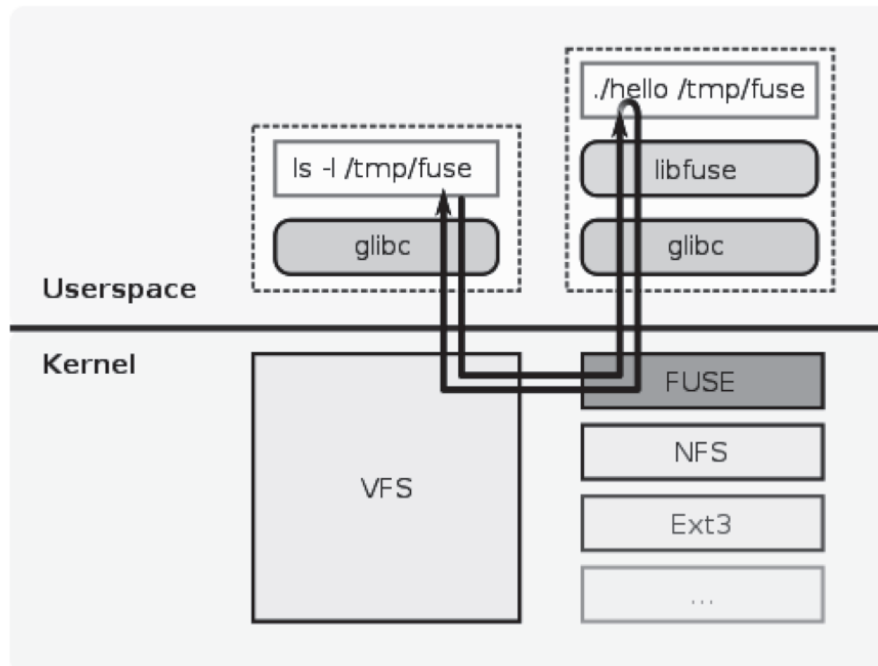

Figure 1: FUSE library architecture

The architecture of Clean Slate consists of the following main components.

- The filesystem
- Key managing service
- Host Authentication Service
- Logging Service

The communication between the server and the filesystem happens via HTTPS in XML format.

Data Access Control

Data Access control is provided to control data access in case the device is stolen/lost. It needs to be designed in such a way that encryption and decryption is transparent to the user. This ensures that even an average user can use the file system without many hassles. We also need to protect the user from him/herself. This is achieved through storing the keys in a remote key managing server. Storing the keys remotely ensures that encryption keys cannot be discovered due to user's ignorance or lack of technical skills and knowledge. This also facilitates blocking access to data by denying access to the encryption keys. Data is encrypted using a symmetric key AES 256 bit algorithm in cipher block chaining mode. We encrypt data, file and directory names including file extensions. This means that the actual file and directory names are not stored anywhere. This raises an issue that how do we know which key on the server belongs to which file.

5. CRYPTOGRAPHIC PROPERTIES

Encryption

We use AES_256 in CBC mode using the openssl library. Openssl is a widely used and tested encryption suite and so is AES_CBC mode. The cipher block size for AES is 16 bytes. If the input plaintext is not a multiple of 16, necessary padding is appended to it to make it a multiple of 16 using the PKCS5 specification. The incoming data to be encrypted is divided into block sizes. The default block size for a fuse filesystem is 4096 bytes. But this can be tweaked depending upon the data that is mostly stored in the filesystem. For example, if the data mostly consists of text document it can be tweaked down to 1024 bytes to allow for more efficient I/O or if it involves mostly multimedia files like videos etc, that usually have a large size, block size can be increased accordingly for that too.

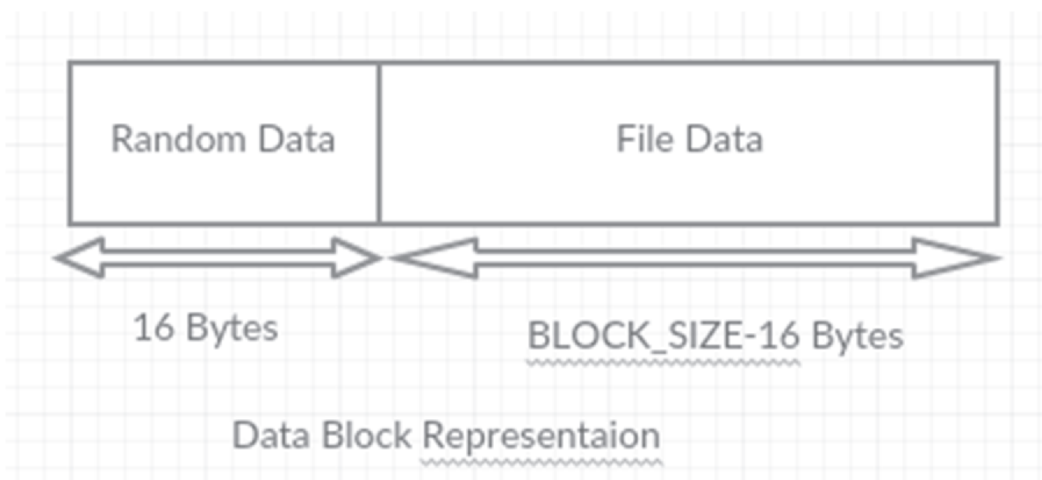

Figure 2: Data block representation

A problem that arises when using CBC mode is the use of IVs. Using the same key+IV to encrypt two messages sharing a common prefix (of one or more blocks) will reveal the presence and length of that prefix, since the ciphertexts will be identical up to the point where the first plaintext difference occurs. Thus facilitating the attacker in retrieval some parts of data. This is especially dangerous when the data in question is textual. Thus we will have to generate a new IV for every new block of data that we have to write. Storing the IVs as well in the remote server will severely downgrade the file system's performance as we will have to transfer IVs to and fro over the network again and again. So, to avoid this, instead of using a unique IV per data block of a single file, we use a single unique IV for the entire file. Then we generate a random 16 byte data block to be used as the first data block.

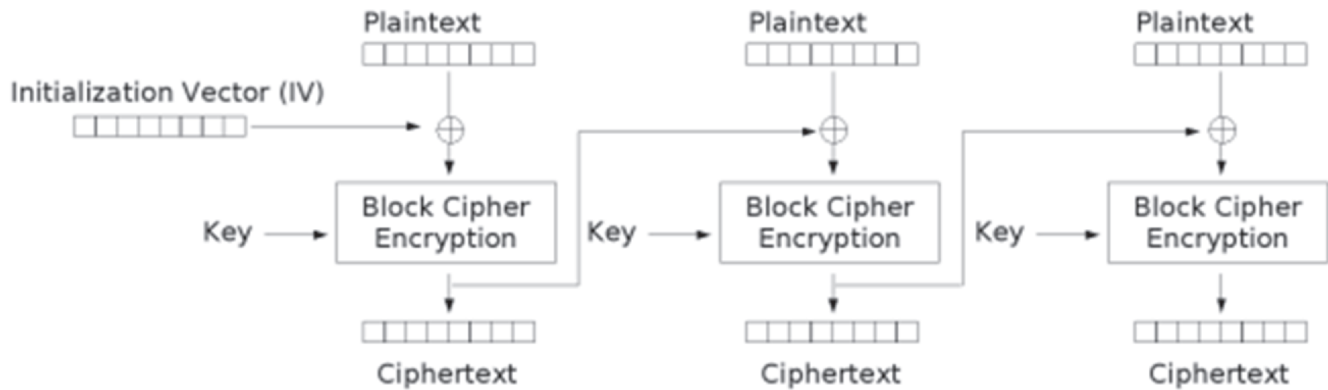

Figure 3: AES CBC architecture

As you can see in the AES_CBC operation diagram above, IV is against the first plaintext block. This encrypted block is then used to encrypt the blocks that follow it. Thus, if the first block itself is random and unique we can guarantee that the issue of same data block and same IV is resolved as we can guarantee that the first data block will be random. This way we overcome the problem of generating and storing different IVs for different blocks within the same file.

The entire process can be summarized as follows:

1. *Csfs_write(path, buf, size, offset, fuse_file_info fi)*
   a. *Remaining_bytes = size+16*
   b. *Prepend random 16 bytes to buffer*
   c. *File_size = get_size(fi->fh)*
   d. *While(remaining_bytes> 0)*
      i. *If(remaining_bytes<=BLOCK_SIZE)*
      ii. *Encrypt(buf)*
      iii. *Write(buf)*
      iv. *Remaining_bytes = 0*
   v. *Return bytes_written*
   e. *Elseif(remaining_bytes>BLOCK_ZIE)*
      i. *E_buf=encrypt(buf[offset],BLOCK_SIZE)*
      ii. *Write(e_buf)*
      iii. *Remaining_byte=Remaining_bytes-BLOCK_SIZE*
2. *Return bytes_written*

Decryption

To decrypt the data, we need the key used to encrypt the data (K_d), the IV (IV_d), used along with it. Both key and IV are unique to the file. K_d, IV_d are all stored in the remote server. To get the required data, we

decrypt a block and remove the first 16 bytes that we prepended during the encryption process, and then return the data to the calling program.

The cryptographic structure can be described as follows:

$$Data = D (E_{data}, Key_{data}, IV\ data)–Random_First_16_Bytes$$

We get following information when a read request is made. File path,offset, and no of bytes to read. Offset%BLOCK_SIZE gives us the block number that we have to start reading from. The number of bytes to be read tells us how many consecutive blocks the data is requested from. Since the data was encrypted a block at a time, it can only be decrypted a block at a time. So we will have to decrypt all blocks corresponding to the data requested as the data can span several complete and a partial block, remove the first random 16 bytes from each block and return the required data to the calling program.

This is achieved as follows:

We first see how many bytes are to be read. Then we determine how many blocks the data spans to by dividing the number of bytes with block size. We find the offset block by dividing the given byte offset with the BLOCK_SIZE. We read the calculated number of blocks, decrypt them and then return the data starting from the offset byte.

The entire process can be summarized as follows:

1. *Csfs_read(path, buf, size, offset, fuse-file_info fi)*
   a. *Pending_bytes = size*
   b. *Offset_block_no = offset/BLOCK_SIZE*
   c. *Block_offset=offset_block_no*BLOCK_SIZE*
   d. *If(size%BLOCK_SIZE == 0)*
      i. *Pending_blocks = size/BLOCK_SIZE*
   e. *Else*
      i. *Pending_blocks=(size/BLOCK_SIZE)+1*
   f. *While(pending_blocks> 0)*
      i. *Read(BLOCK_SIZE bytes)*
      ii. *Decrypt(data)*
      iii. *Remove_First_16_Bytes*
      iv. *Store data*
2. *Return plaintext_data*

6. HOSTAUTHENTICATION

For host authentication, we need a mechanism that is both light on the resources and secure. For this, we propose using digital certificates. The host authentication service is responsible for handling these operations and also acts as the Certificate Authority. When the device is first set up with clean slate, a certificate sign request is issued by the host system. The authentication service generates a digital certificate by signing it with its private key. This certificate is then stored in the platforms certificate store (Windows Certificate Store and /etc/ssl/certs in linux). Now every time that the filesystem is to be mounted, the remote authentication service asks for the certificate, if the certificate is valid, filesystem is mounted, if

the certificate is invalid, or not present, mounting is denied and the storage device is considered as lost or stolen. Along with this an alert is sent to the user through email or SMS that a failed attempt on data access was made. The SMS or email can also contain the address of the host that failed authentication if possible.

Usage of certificates also makes it possible to use the same device on multiple hosts by exporting the digital certificate to multiple hosts.

Certificate content:

- Serial Number
- Certificate algorithm identifier
- Valid After
- Device's public key information
- Digital signature made with host's private key.

For mobile phones the same can be used in addition to the change of the SIM card.

Logging and Alerts

Encryption like every other technology isn't perfect, and can be compromised if the attacker is motivated enough or some other unforeseen vulnerability in the encryption scheme exists. Assuming an attacker is able to circumvent all the above security measures somehow, and gains access to data, we mean to able to identify that as we wouldn't like to operate under an illusion that the security has not been compromised when it really has. Hence for this we propose a logging and alert system, which will log file system activities before they are performed, such that no operation can take place without it getting logged first. This logging system will provide us with crucial information like which particular files have been compromised, the IP address of the machine that attempted to access data etc. The logging data needs be precise containing only the needed information, like file that was accessed, time of access, and the IP address used to access the file. Along with this, an alert is sent to the user via an email or SMS whenever the filesystem is mounted. So that even when the user isn't aware of the theft or loss immediately, he/she can quickly take countermeasures such as report of the theft and locking the access to the keys. When we know what data may have been compromised we are able to take better informed decisions for damage control.

7. PERFORMANCE IMPROVEMENTS

Key Prefetching

Fetching a key for a single file from the server introduces a significant overhead on file system's read performance. Both network latency and bandwidth limitations impact our read performance. Thus we need to reduce the number of requests we make to the server during read operations. This means we have to get multiple keys in a single request, rather than a request per file. But this opens us to main memory read attack vectors like cold boot attacks. If we have too many keys in the main memory, an attacker might be able to get his/her hands on them. Also, what if the attacker steals a laptop with an authenticated session, the attacker will be able to access files that already have their keys present in the cache. We thus, need to keep just enough keys that improve our I/O performance while not severely impacting the security. For this we propose a key prefetching algorithm that uses a weighted directed graph and maximum heap.

Figure 4: Example directory structure

The algorithm can be described as follows. The above figure represents a directory structure within the filesystem.

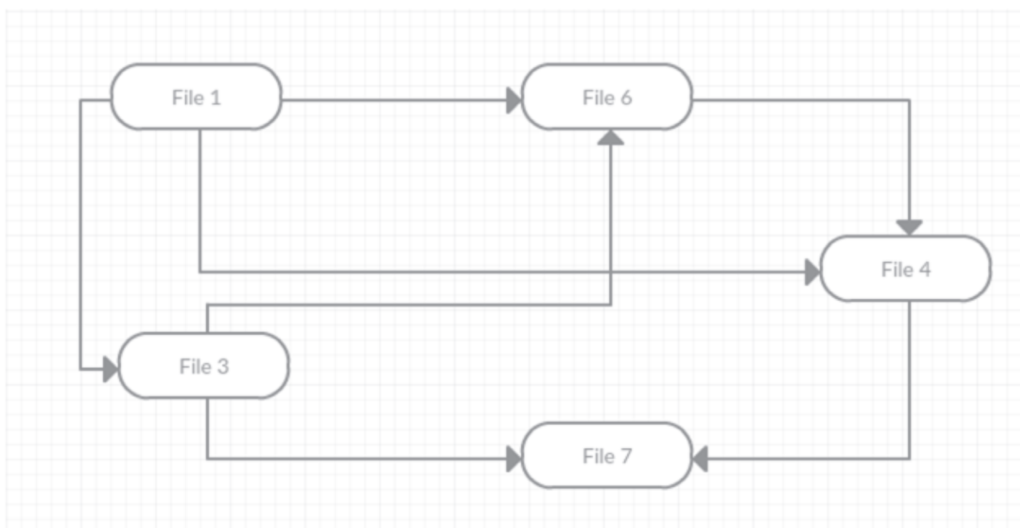

Figure 5: Prefetching graph visualization

The files are considered as nodes in a graph, the edges bear a weight, which is the frequency the file has been accessed with i.e. the frequency with which the particular edge has been traversed in the past. We then prefetch the key corresponding to the destination node of the edge. A modified breadth first traversal algorithm is used. We also define the depth and breadth of prefetching/traversal.

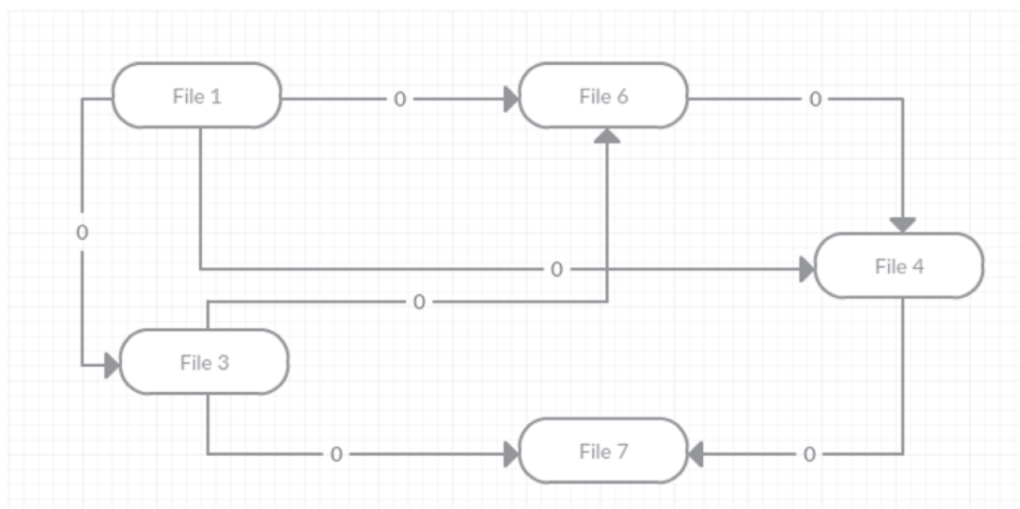

Figure 6: Initial graph edges before filesystem usage

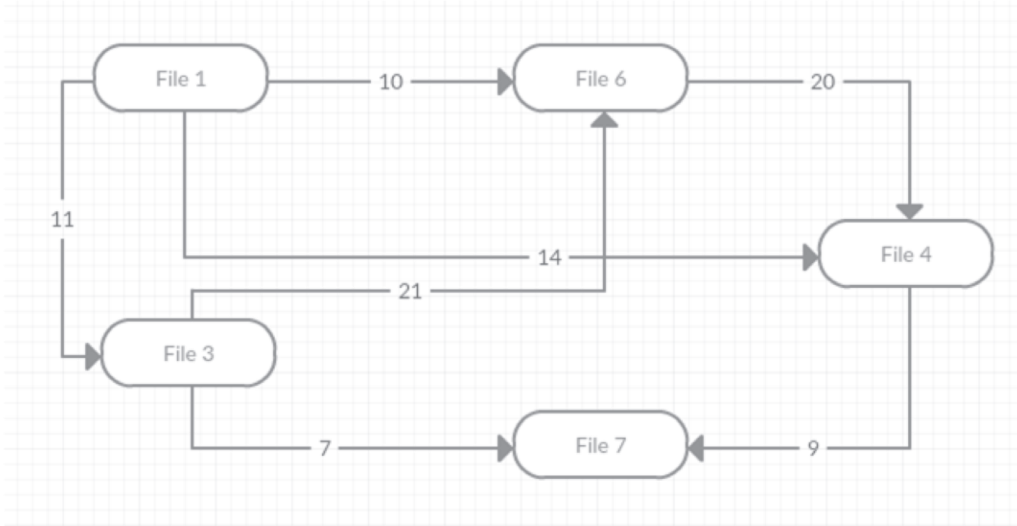

Figure 7: Graph edges after some usage

The prefetching graph G (V, E) can be described as follows:

V → Set of all file ids in the filesystem

E ($f_i f_j$) → Set of all edges connecting two nodes. f_i being the source node and f_j the destination node.

All edges as mentioned before are assigned a weight with is initially 0, and is incremented every time an open request is issued on the file. We then apply a modified breadth first traversal using the heap based priority queue to get the maximum weight edges, hence, traversing the longest path instead of the shortest path.

This graph data, is stored locally on the system, and is encrypted to prevent the attacker from guessing the possible importance of the encrypted files. As this structure can thoroughly reveal the usage pattern of encrypted files. The key for this structure is also stored remotely. The cache as well is encrypted and created only after the file system has been mounted during initialization. It is destroyed immediately afterwards.

Assuming we are coming from file with id n1 we will be working with n1's adjacency list which is a max heap priority queue.

1. *on open(file id n1, file id n2) request*
   a. *if file id n1 has file id n2 in it's adjacency list*
      i. *increase n1[n2[weight]] by 1*
   b. *else*
      i. *insert n2 in adjacency list*
      ii. *initialize n1->n2's weight to 0*
   c. *fetch keys(int breadth, int width)*
   d. *update key_cache()*
2. *during read(file id n2)*
   a. *if(key_present_in_cache)*
      i. *read_data()*
      ii. *decrypt_data()*

b. else

i. fetch_key()

ii. read_data()

iii. decrypt_data()

c. return data

Replacement Policy

Considering the probable size of the key cache which cann range from 5-20, we require an algorithm that is easy to implement and also effective. The CLOCK replacement policy seems ideal for our needs. It's easy to implement as it uses only a circular queue and a single pointer (the hand). This makes it ideal for use where there isn't too much of data to process through. It is based on the Last Recently Used policy.

The circular list contains bits for the cache indices (0, 1).When the cache becomes full and there's more incoming keys, it checks for the bit that is under the hand currently, if it is 0, the particular index in the cache is emptied and a new key is stored. If it is one, it is set to 0 and the hand is incremented onto the next index.

8. CONCLUSION

Clean slate understands the need of normal user's data safety measures, privacy and makes an attempt at making security available and usable by all no matter the level of technical expertise. The solution works at the heart of the problem i.e. data management by implementing these security measures at the file system level. Data access control is achieved by storing keys remotely and providing access only when the host is authenticated. Device pairing is achieved via certificate authentication of the host machine. The logging system helps make informed decisions when the above security measures have been breached. These security measures do not ensure data retrieval after loss/theft but they do attempt at ensuring that no unauthorised access should be possible in such an occasion. The storage of secret keys on a remote server impacts I/O performance, and hence the filesystem cannot be used where performance is a requirement. But the filesystem is usable and the addition of the proposed prefetching algorithm can improve performance. Performance in an informal, real life scenario was usable without excessive delays.

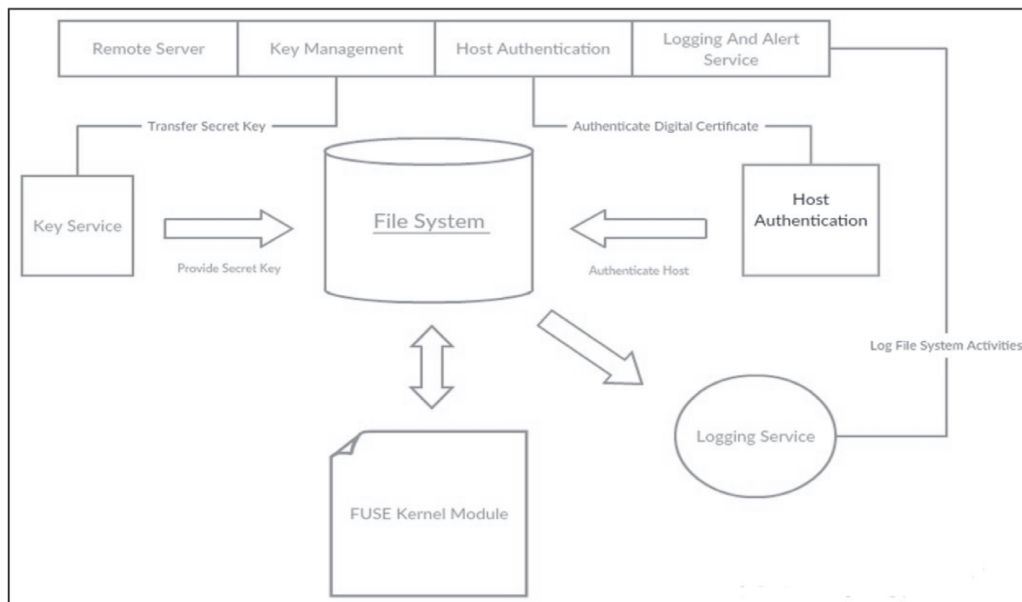

Figure 8: Clean Slate file system architecture

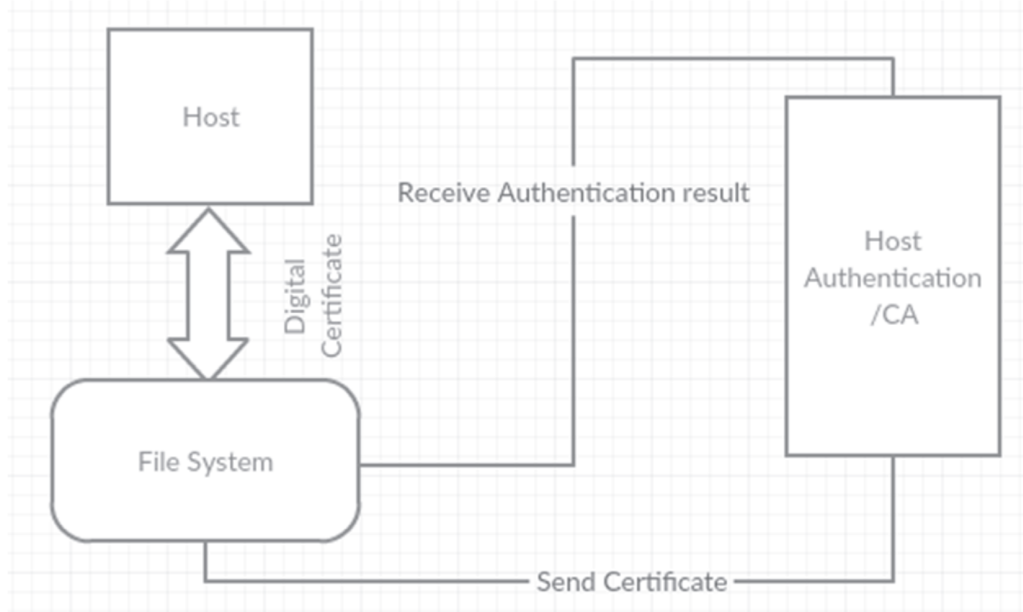

Figure 9: Host authentication mechanism

References

1. Czeskis, D. Hilaire, K. Koscher, S. Gribble, T. Kohno and B. Schneier, '*Defeating encrypted and deniable file systems: TrueCrypt v5.1a and the case of the tattling OS and applications*', Proceedings of the 3rd conference on Hot topics in security, p. 7, 2008.
2. Studer and A. Perrig, '*Mobile user location-specific encryption (MULE)*', Proceedings of the third ACM conference on Wireless network security - WiSec '10, pp. 151-162, 2010.
3. Fuse documentation and source hosting page: https://github.com/libfuse/libfuse
4. Whitten and J. Tygar, '*Why Johnny can't encrypt: a usability evaluation of PGP 5.0*', Proceedings of the 8th conference on USENIX Security Symposium - Volume 8, pp. 14-14, 1999.
5. GlobalSign White Paper: GlobalSign Authentication Achieving a comprehensive information security strategy using certificate based Network Authentication by John B Harris, Security Specialist On behalf of GMO GlobalSign Ltd.
6. Identity Management: Consumer's Habits & the Potential Backlash Faced by Business: Winmark Research (commissioned by RSA Security)April2004. URL: http://www.rsasecurity.com/go/ntk/idmreport/IDManagement.pdf
7. Kukec, S. Gros and V. Glavinic, '*Implementation of Certificate Based Authentication in IKEv2 Protocol*', 2007 29th International Conference on Information Technology Interfaces, pp. 697 - 702, 2007.
8. Micro-Trax Statistics: http://micro-trax.com/statistics.
9. S. Sharma, R. Moona and D. Sanghi, '*Transcrypt: A Secure and Transparent Encrypting File System for Enterprises*' Proceedings of the 8th International Symposium on Systems and Information Security (SSI 2006), Sao Paulo, Brazil, November 8-10, 2006.
10. W. Wen, T. Saito and F. Mizoguchi, '*Security of Public Key Certificate Based Authentication Protocols*', Public Key Cryptography, pp. 196-209, 2000.