Finding the Right Balance: Security vs. Performance with Network Storage Systems

A Thesis Presented

by

Arun Olappamanna Vasudevan

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

Technical Report FSL-15-01

May 2015

Stony Brook University

The Graduate School

Arun Olappamanna Vasudevan

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Associate Professor, Computer Science

Dr. Scott Stoller, Thesis Committee Chair
Professor, Computer Science

Dr. Mike Ferdman
Assistant Professor, Computer Science

This thesis is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Thesis

Finding the Right Balance: Security vs. Performance with Network Storage Systems

by

Arun Olappamanna Vasudevan

Master of Science

in

Computer Science

Stony Brook University

2015

As cloud storage in particular and networked storage in general gain widespread adoption, the biggest concern for customers is security. The concern is well warranted for two reasons. First, the surface area of exploitation and vulnerability is greatly increased with the communication channel and a possibly shared remote server, in addition to multiple clients. Second, when the networked storage server is maintained by a third party, such as a cloud provider, there is a lack of trust on the way data is managed at the server side.

In order to minimize the risks while out-sourcing the data management, networked storage clients need security mechanisms at their end to ensure data integrity and confidentiality. However, security mechanisms such as encryption, authentication, and virus-scanning often have high performance overhead. Figuring out the security policy that offers the right balance of security and performance is therefore important. To solve this problem, we examine the performance overhead of different security features of a networked storage system, and develop security policies that trade-off security and performance. This study was motivated by the lack of previous work on performance overhead of security in the context of networked storage systems.

A typical way to enforce security policies in client-server computer systems is using proxies to monitor and regulate the client-server communication, as exemplified by network firewalls. While the security installations go into a proxy, clients and servers are kept intact, and can continue to work without changes. The proxies are usually deployed at the trusted end to fence off security threats from the untrusted end. For example, network firewalls are often deployed by the server end to defend malicious clients; cloud-backed storage proxies can be used by the client end to safeguard against the malicious cloud back-end. In addition to providing security, proxies can also improve performance by caching as with CDNs. Considering that storage servers may be slow and over a WAN, as in cloud-backed systems, caching proxies, deployed in the same LAN of the clients, can significantly reduce server access latency.

Specifically, we studied the trade-off between security and performance in a Network File System (NFSv4) with a security and caching proxy. We designed and implemented the proxy with a layered architecture, where each security feature is a stackable file system layer. Each layer can be enabled or disabled, and configured separately as required by policy. For example, an anti-virus layer can be configured with the size or type of file that it scans, while an integrity layer can be configured independently whether or not to detect replay attacks of file data. This layered architecture facilitates the security-performance trade-off study because different security policies can be composed easily via composition of different layers.

Our study showed interesting interaction between security policies and system performance. We found that the order of the same set of security layers has significant performance impact, and identified the optimal order of anti-virus, encryption, integrity, and caching layers. In addition to a broad idea of security policies and their effect on performance, we also present insight into interactions between caching and security, a topic that is less studied in academia.

To my Mother Usha;
Father Vasudevan;
Teachers Smt Shobhana, and Sri Krishnamurthy;
Grandparents Muthassan, Muthassiamma, and Ammamma;
and
Sri Krishna, the Divine, the Consciousness!

Contents

List of Figures

List of Tables

Acknowledgments

“The whole is greater than the sum of its parts,” – Aristotle.

This work could not have been possible without the efforts and sacrifices of a lot of people.

Dr. Erez Zadok has been extremely supportive throughout my spell at the File Systems and Storage Lab (FSL). He was very involved with this work and gave invaluable suggestions. In fact, the very idea of studying the interaction between cache and security modules was his. I express my heartfelt gratitude to Prof. Zadok.

Ming Chen, a Ph.D. student at FSL, helped me with the project in several ways. It is amazing that he was able to offer immensely to this project with coding, debugging, reviewing, benchmarking, and project management. He did all of this on top of his thesis work and personal responsibilities. He is such an inspiration, and I thank him profusely for everything.

Kelong Wang, a masters student at FSL, also worked hard to get a lot of work done in relatively less time. I got to learn several concepts from him. Thankfulness to Kelong for his contribution.

I also acknowledge all the other students at FSL. We spent quality time with each other and discovered a lot.

Let me also appreciate Prof. Mike Ferdman (who taught Operating Systems and Computer Architecture courses), Prof. Jie Gao (Algorithms), Prof. R. Sekar (Compilers), Prof. Don Porter (Virtualization), and Prof. Long Lu (System Security). Courses with them improved my knowledge of systems. Toshiba’s compiler division in Bangalore, who I worked with, laid the foundation for my understanding of systems. I extend a special recognition to Balachandher Sambasivam and the team.

Finally, gratefulness to the valuable feedback by committee members – Dr. Scott Stoller, Committee Chair; Dr. Mike Ferdman; and Dr. Erez Zadok, Thesis Adviser.

This work was made possible in part thanks to NSF awards CNS-1302246, CNS-1305360, CNS-1522834, and IIS-1251137.

This journey has not been a solitary one. My family and friends have ridden along!

Chapter 1

Introduction

Cloud storage is popular as we can see from the prosperity of a long list of cloud storage providers, such as Amazon S3, Google Drive, Dropbox, Microsoft OneDrive and Azure, Apple iCloud drive, Box.net, etc. It is becoming increasingly popular as the vision of utility computing is gradually being realized. More organizations and people have moved their enterprise and personal data to the cloud as an economically more viable alternative to local storage for traits such as data protection, scalability, and accessibility (from multiple devices, at multiple locations) [5, 27]. Table 1.1 shows impressive availability statistics from major cloud providers [45].

Outsourcing data storage to third-party cloud providers is a challenging decision with the following concerns:

Availability Data should be available when needed.

Confidentiality Data should be accessible only to authorized users.

Integrity Data read should be same as data that was written.

While cloud storage has great availability, it is far from perfect. Data loss continues to be a concern. A 2013 report by Symantec found that 43% of respondents have lost data in the cloud [16]. Data loss prevention strategies include auditing the cloud, creating copies, keeping data across multiple cloud providers [6, 10, 23, 25], etc.

Cloud Providers	Storage downtime (hours)	Storage uptime %
Microsoft Azure	10.97	99.8751
Amazon Web Services	2.69	99.9694
Google Cloud Platform	0.23 (14 min)	99.9973

Table 1.1: Storage service downtime of major cloud providers in 2014

In terms of integrity and confidentiality, cloud storage is doing much worse. It is scary to note that there are reports of data corruption in the cloud [42], since it could easily go undetected for long time, and used by applications as legitimate data. For instance, the data in financial institutions, if corrupt, can lead to all sorts of incorrect calculations that affect a lot of customers. With increasing volume of private data in the cloud, privacy and confidentiality are big concerns. The significance is highlighted by highly publicized incidents such as leakage of intimate photos of celebrities [4], stealing of patient records [30], and release of confidential data of companies [44].

Security issues are inherent in cloud's nature that data are managed be service providers in a *cloudy* (opaque) manner. This *cloudiness* is flexible so that providers can consolidate, scale, replicate, or migrate data without affecting clients. Meanwhile, the *cloudiness* also begets mistrust of cloud storage providers. Even if the providers themselves are trusted, security safeguards are still necessary because many cloud-storage providers share physical resources among multiple potentially-malicious tenants.

In this thesis, we study how to secure outsourced data while assuming availability as the bare minimum requirement that clients depend on the server for. The assumption is reasonable because high availability is one of the key factors that motivate the adoption of cloud storage. We focus on means that clients can use to ensure the confidentiality and integrity of data. Confidentiality can be fully achieved through encryption of data that goes into the cloud, whereas integrity can be verified by checking retrieved data against Message Authentication Code (MAC). Security concerns about the Internet that connects the organization with the cloud storage is significant, but has been greatly reduced by HTTPS or Kerberos [41] that are used for transport-layer security. Although a trusted transport layer is desired, our study of integrity and confidentiality is agnostic to and does not require a trusted transport.

As an organization that uses a third-party provided cloud service, encryption and integrity-checks are enough to take care of server-side trust issues. But, the end-users of cloud storage facility in the organization, for instance, the employees, cannot be fully trusted. There could be accidental infusion of virus files into client systems. Thus an organization should guard against malware as well. Additionally there could be other issues to guard against, such as accidental modification or deletion by the end-users. Solutions such as snapshotting can be deployed to protect against this.

In this thesis, we focus on confidentiality and integrity issues with the network storage server, and malware concerns at the client-side. As already discussed, the solutions are to deploy data security measures such as encryption, integrity-checking, and anti-virus. The security solutions often come with a performance overhead. It is important that we understand and characterize the performance overhead and trade-offs of security, so that the right decisions can be made when determining security policies. For example, we might be able skip some security measures for certain temporary files. There could also be personal files such as photos that may not be all that relevant for the company to protect with a full suite of security measures.

To our best knowledge, there has not been any prior work that studies the security–performance relation in a networked storage system. This work will serve as a good reference for future research in this direction, in addition to helping decide security policies for similar systems.

With an untrusted server and a semi-trusted client, using a proxy or a gateway machine is a good way to monitor and regulate the client-server communication, as shown in Figure 1.1. The proxy machine, being under the control of system administrators, is trusted and can guard

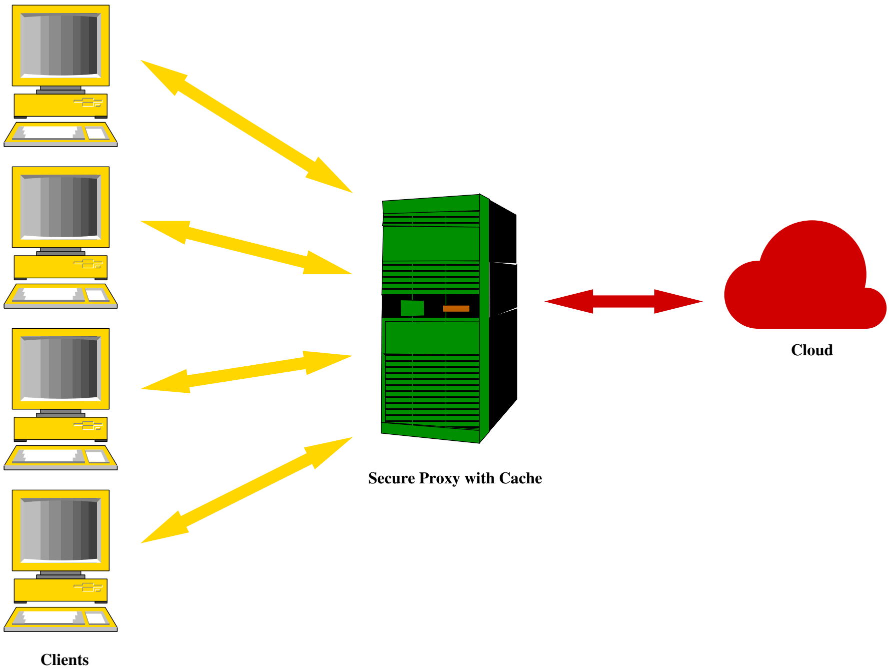


Figure 1.1: Secure proxy

against security threats from both untrusted servers and semi-trusted clients. The proxy also allows the client and server to continue to work as they used to, with minor modifications such as the IP address to connect to. The proxy machine can be used to perform operations to find checksum, encrypt, and scan for malware, when data is written to server; and decrypt and verify checksum when data is read from server.

However, security is not free. Some researchers [8] argued that encryption was too expensive to justify storing encrypted data on the cloud, whereas other researchers [1,43] have claimed new hard acceleration make encryption viable and cheap for cloud storage. The overhead of security might be minimized by caching contents and collecting multiple writes before writing back to server. The benefits of a cache could turn out to be significant, considering that the storage server is accessed over the long-latency Internet. The proxy, on the other hand, is in the same LAN as client machines.

The system that we built has a client that accesses data on a storage server using a Network File System (NFS). Though in reality, a cloud storage would typically have a REST or SOAP API [3], we selected NFS as the protocol for mainly two reasons. First, NFS version 4 is designed for WAN with features such as a standard port number (2049), stronger network security using RPCSEC_GSS, compound requests, and delegations [7]. The newer version 4.1 supports parallelism and data striping for improved performance. Second, there are systems like Panache from IBM [26] that enable wide-area file access through caching.

With clients and the server using NFS, the proxy acts as NFS server to client and as an NFS client to the server. We developed security features and caching as layers on top of the NFS client part of the proxy. Development in stackable file system layers enabled easier development and debugging. The stackable architecture also allowed enabling, disabling, and even reordering of the layers, to compose different security policies and to study the effect of each layer separately. In fact, we found that the order of the same set of security layers has significant performance impact, and identified the optimal order of anti-virus, encryption, integrity-check, and caching layers.

The rest of this thesis is organized as follows. Chapter 2 discusses background work to justify the need for analyzing the impact of security policies on performance. Chapter 3 talks about the interaction between security and caching modules from a theoretical perspective. Chapter 4 details our implementation. Chapter 5 compares the performances of different security policies using micro and macro workloads. This section also summarizes the correctness of our implementation. Chapter 6 concludes and discusses future work.

Chapter 2

Background

Though our work to find overhead of security in networked storage systems is novel, there has been similar work in networking, earlier. In this section, we talk about HTTPi and related technology that gives a choices of trading security with performance in networking space. We follow this up with a few network storage systems that could use our research work. This chapter ends with a brief about NFS-Ganesha and FSAL that is used in the system.

2.1 Prior Work

2.1.1 HTTP with Integrity

In the World Wide Web, two common transfer protocols are HTTP and HTTPS. While HTTP gives no security, HTTPS provides security guarantees for server authentication, message integrity, and message confidentiality. HTTPS achieves confidentiality by encrypting messages using a symmetric key that is exchanged at the time of establishing connection. The exchange of symmetric key is protected by asymmetric encryption using public and private keys. Encryption of messages have a two-fold overhead. First, each message has to be encrypted and decrypted, that adds to computation overhead. Second, the message cannot be cached in intermediate servers, that adds to bandwidth requirements and network traffic.

In several cases such as open applications, server authentication and message integrity are the only security guarantees that are needed and using HTTPS is an overkill. This led to proposal of HTTPi (HTTP with Integrity) [9, 38] and SINE [19]. A list of HTTP-related protocols and their security guarantees are listed in Table 2.1 [9]. With HTTPI, intermediate servers can cache contents since the content is irrespective of which client accesses it. This also enables the use of HTTPI in Content-Centric Networks and Named Data Networks [22]. HTTPI's performance is almost as good as HTTP.

In case of networked storage systems, unlike the World Wide Web, server authentication is not a problem, whereas, client authentication, message integrity, and message confidentiality are. The emergence of newer protocols to satisfy specific security guarantees in the World Wide Web is a testimony and motivation to our study of performance overheads of different security policies in a networked storage system. Apart from the motivation part, the HTTP family bears no direct relation with the research that this thesis is about.

	Server Authentication	Client Authentication	Message Integrity	Message Confidentiality
HTTP				
HTTPS	✔	✘	✔	✔
HTTPS with password	✔	✔	✔	✔
HTTPA [17]	✘	✔	✔	✘
HTTPI	✔	✘	✔	✘
HTTPI with password	✔	✔	✔	✘

Table 2.1: Security guarantees of HTTP, HTTPS, HTTPA, HTTPI

2.2 Network Attached Storage and Cloud

Network Attached Storage (NAS) appliances are popular as a storage solution for enterprise. There are several storage companies that provide NAS solutions with a lot of features. For instance, FreeNAS [21], a free and open-source software for NAS, comes snapshotting, replication, compression, disk encryption, de-duplication, etc. NAS provides file-based protocols such as NFS and CIFS for access by clients. If security of this system has to be enhanced to detect malware, for example, we can use anti-virus scanners before writing to storage server.

Our work can give an idea of performance overhead for security enhancement of NAS appliance. We have noted down a few more examples where our research can be used, in the next few paragraphs.

2.2.1 IBM Panache

Panache [26] is a parallel file system cache that enables efficient global file access over WAN without the fluctuations and latencies that WAN comes with. It uses pNFS to read data from remote servers over the Internet and caches them locally in a cache cluster. Our work will prove beneficial if IBM Panache is to be fortified with security enforcements. In that case, the results of our study can help Pananche decides the security features to pursue considering the potential performance overhead.

2.2.2 Cloud NAS

Virtual NAS on cloud such as Amazon S3 and Microsoft Azure is provided by companies like SoftNAS [39] and Zadara Storage [47]. The clients to these cloud NAS servers are the cloud compute nodes provided by the same provider. Thus there could be Amazon EC2 instances accessing a SoftNAS composed of Amazon S3 storage nodes. In such scenarios, if we enhance the security of the system, our methodology of studying security features' performance overhead can also be applied to SoftNAS.

2.2.3 Cloud storage Gateways

A cloud gateway appliance or virtual machine typically gives a transparent SAN or NAS interface for local applications. On the other side, the cloud gateway appliance accesses cloud storage by translating requests from local applications to a REST API. The motive for a cloud storage gateway need not be necessarily for a protocol translations. There are cloud gateways that provide one or more of features like deduplication, replication, security and caching are also available.

There are several cloud gateway technologies, in both industry and academia. Hybris [15], BlueSky [43], and Iris [40] are examples of cloud storage gateway systems that provide integrity. Hybris additionally gives fault tolerance by using multiple cloud providers. BlueSky provides encryption. BlueSky and Iris have a file system interface on the client side, and Hybris provides a key-value store. In the industry, NetApp SteelStore [31] is a cloud integrated storage for backup. The exact security features in SteelStore are not known. Our work on study of security overhead can be extended to cloud storage gateways to fine tune security policy.

2.3 NFS-Ganesha

NFS-Ganesha [12, 13, 18] is a user-land implementation of NFS server. NFS-Ganesha can be configured to export local file system or act as proxy for another NFS server. While NFS Ganesha supports NFS v3, v4, and v4.1, the proxy configuration works only with NFS v4 onward.

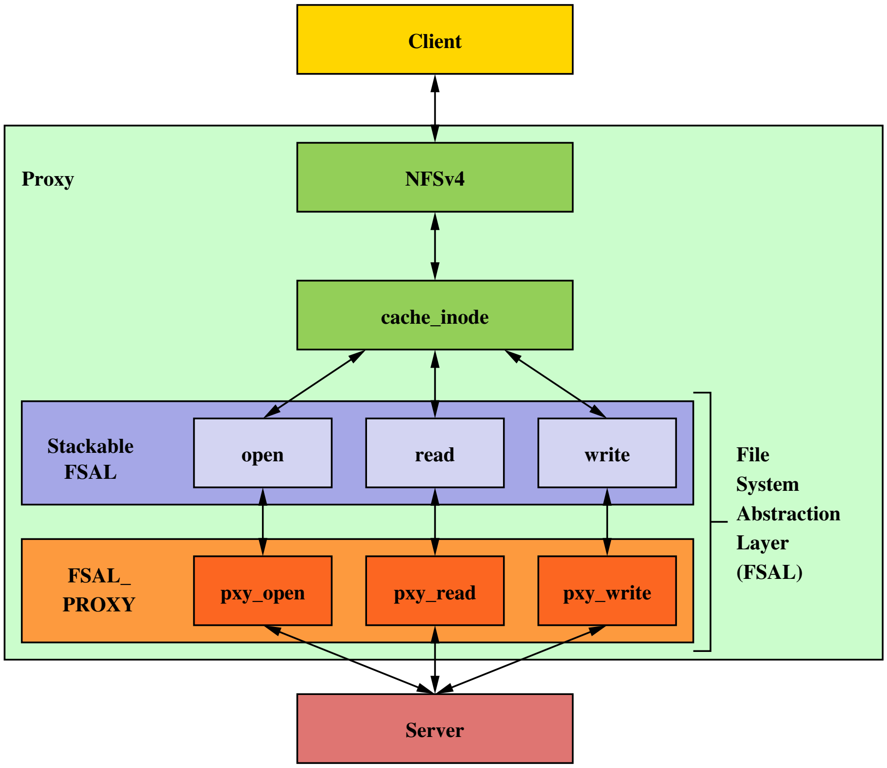


Figure 2.1: NFS-Ganesha in proxy configuration – overview

Figure 2.1 shows a high-level overview of NFS-Ganesha in proxy configuration. The config-parser module creates exports at the time of initialization. Once up and running, NFS requests from clients are handled by the NFSv4 module. The cache-inode module is a meta-data cache layer, and it also deals with content and attribute lock corresponding to each file. The cache-inode module interacts with underlying file system via an abstracted interface called File System Abstraction Layer (FSAL). FSALs are much like VFS in Linux, and provides a generic interface to multiple file system implementations. Each FSAL has a vector of functions for FSAL life-cycle management; file operations like open, read, write; and name-space operations like lookup, create, makedir.

FSALs can also be stacked to give a more features in a modularized manner. For example, FSALs for caching, encryption, integrity-checking, and anti-virus can be stacked on top of the FSAL_PROXY. Stackable FSALs are similar to stackable file systems in Linux [49]. In the Figure 2.1 below, we have a caching FSAL, the FSAL_PCACHE, above FSAL_PROXY. The original implementation had support for one stackable FSAL on top of FSAL proxy. We contributed to NFS-Ganesha to support multiple stack [34].

The way cache inode module interacts with underlying FSAL is using a `fsal_obj_handle`, which is a data structure similar to inode in VFS in Linux. Typically, every FSAL layer maintains its own `fsal_obj_handle` and keeps a pointer to the next layer's object.

Each FSAL module is built as a shared object library that gets loaded based on FSAL listed in export configuration.

Chapter 3

Design

In this section, we talk about the threat model, design goals, and the architecture of our secure network storage proxy.

3.1 Threat Model

Our threat model reflects the settings of an enterprise office or academic laboratory where multiple clients access the data on a storage server. In the case that the storage server is outsourced to the cloud, the server is untrusted, and the clients are semi-trusted. The three main information security requirements are confidentiality, integrity, and availability. We discuss the trust on server and client with regard to these.

Server. Storage server is expected to have reasonably high up-time. Hence, availability is not a problem. However, when it comes to confidentiality and integrity, the server is not trusted at all.

Client. Clients are semi-trusted. That is, while the clients are not per se untrusted, there could be accidental intrusion of malware that compromises the clients.

Since both the server and clients are not fully trusted, the solution is to use a trusted proxy, as mentioned in Chapter 1. With a fully trusted proxy, all security enforcements and data monitoring can be done in the proxy. In this study, we consider the simple case that there is just a single proxy that is connected to multiple clients and a single server.

3.2 Design Goals

The intent of this thesis is to measure and understand the overhead of different security policies in networked storage systems. In order to measure the overhead of different security policies in networked storage systems, we need a security architecture that

- supports advanced security features for integrity, confidentiality and availability of data;

- allows configuration of security features to compose different security policies;
- is compatible with existing system;
- minimizes impact on performance; and
- is easy to develop and maintain.

Security is enforced in a proxy machine that sits between clients and storage server. This way, clients and server can work without upgrades. With proxy in the same LAN as clients, a cache in proxy can even offset the overhead of security enforcement to some extent. The security and caching modules are implemented in a layered architecture. This helps easier development and maintenance of each security layer independent of other layers. Moreover, each security layer can be either enabled or disabled, based on security policies.

The proxy and layered architectures are discussed in detail in upcoming section.

3.3 Security and Caching

In a layered architecture, the order of the layers has important performance implications. We first discuss a simplified secnario with only one generic security module and one cache module and try to see how they can be stacked in a proxy machine.

There are two ways of stacking a cache module and a generic security module, as shown in Figure 3.1.

A security module logically closer to client and

B cache module logically closer to client

To get further insights into these two approaches, we studied them using a general I/O model, which describes common file system operations using simpler key value operations.

3.3.1 A General I/O Model

The terminology used in describing the model is as follows. "Stack" or "layer" in proxy have to adhere to the server–client access protocol. "Module" is internal to a layer and can have its own interface. Though layers have to adhere to server–client access protocol, there is no restriction regarding exposing of additional functions. For example, there could be special interactions between cache layer and security layer.

In our general I/O model, we assume a key–value store in the server. There are only 3 operations to access the store - read, write, and remove. While read and remove perform what the name suggests, write has an additional functionality of creating a new key–value pair, if the key does not exist. Since key creation is not part of read and remove, they could return error code NOT_FOUND if the key is not found in the server. Read operation may additionally return INVALID code if the key is found but the value is not secure.

To define the model more formally, we use the following types:

- `key`

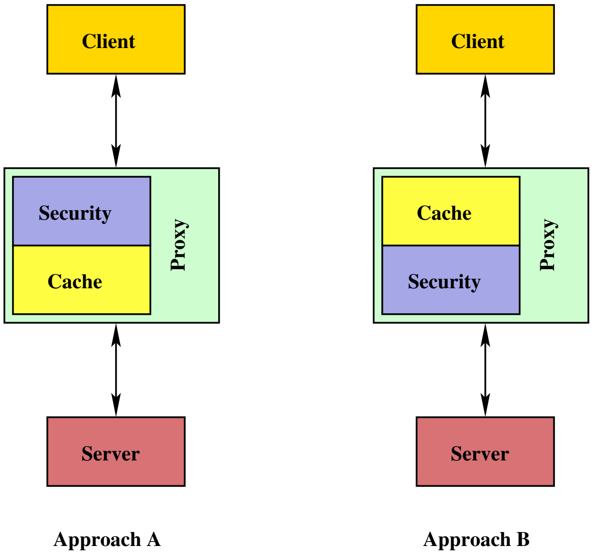


Figure 3.1: There are two ways that a security and cache module can be stacked in a trusted proxy

- `value`

-
  ```
  typedef struct {
    value val;
    int   err_code;
  } ret_t;
  ```

The key is a tuple of file handle, offset, and length; the value is file data. File system operations described using the model are listed in Table 3.1.

Simple Cache Module

The simple cache is a module that uses the same interface as in Table 3.1. Cache maintains a local key–value store. The local key–value store has to maintain additional meta-data, for instance, to know whether value corresponding to a given key is clean or dirty.

The operations for accessing the key–value store is very similar to the server access interface in Table 3.1, except for an additional meta-data field update in case of write. Also, to differentiate from the client-server interface described earlier, we use `insert` for writing to the cache, `lookup` for reading from cache, and `delete` for removing a key from cache. These operations are listed in Table 3.2. We need a new struct for meta-data:

- `meta-data`

-
  ```
  typedef struct {
    value     val;
    meta-data meta;
    bool      status;
  } cache_ret_t;
  ```

Operation	**Parameters**	**Return value**
`ret_t read(key k);` Read value corresponding to key `k`	`k` – key for which value needs to be retrieved	`.val` – value corresponding to key if `.err_code` is `FOUND` `.err_code` – `FOUND` if key `k` is found and value corresponding to key is valid `NOT_FOUND` if key `k` is not found `INVALID` if key `k` is found but corresponding value is not valid. For instance, if the value is found to be malicious, then this error code is returned.
`bool write(key k, value v);` Write value `v` for key `k`. If key `k` does not exist, a new key–value pair is created. If key `k` exists, its value is rewritten with `v`. There is a possibility that the write fails.	`k` – key for which write is done `v` – value corresponding to key `k`	`TRUE` if write succeeds `FALSE` if write fails. For instance, if the value is found to be malicious or due to space limitations.
`bool remove(key k);` Remove key–value corresponding to key `k`	`k` – key that should be removed	`TRUE` if key exists `FALSE` if key does not exist

Table 3.1: Simple key–value storage access protocol

Operation	**Parameters**	**Return value**

`cache_ret_t lookup(key k);`
`cache_ret_t check_flags(key k);`
Lookup for key `k` and return value, meta-data if found. While `lookup` returns both value and meta-data, `(`check_flags`)` returns just meta-data.
	`k` – key that should be looked up	`.val` – value corresponding to key if `.status` is `TRUE` and function if `lookup`
		`.meta` – meta-data corresponding to key `k`
		`.status` – `TRUE` if key `k` is found
		`FALSE` if key `k` is not found

`bool insert(key k, value v, meta-data m);`
`bool insert(key k, value v);`
`bool mark(key k, meta-data m);`
`void mark_all(meta-data m);`
Insert to cache value `v`, if any, and meta-data `m`, if any, for key `k`. For mark_all, update meta-data `m` for all cached keys. If key `k` does not exist, a new key–value pair is created. If key `k` exists, its value is rewritten with `v`.
	`k` – key for which insert is done
	`v` – value corresponding to key `k`
	`m` – meta-data corresponding to ket `k`
		`TRUE` if insert succeeds
		`FALSE` if insert fails.

`bool delete(key k);` Remove key–value–meta-data corresponding to key `k`	`k` – key that should be removed	`TRUE` if key exists `FALSE` if key does not exist

Table 3.2: Simple cache interface

Simple Security Module

The role of security module is to check if a given value is "good" or "bad." In order to check the security of a given value, additional values might be needed. For example, if a non-aligned offset in a file is accessed, an encryption module needs the entire block. This is a case where before doing any computations for security check, the dependency is identified. An anti-virus module on the other hand, may want to check more data, based on pattern detected in the currently accessed data. This is a case where during the security check, requirement of more values is realized.

The operations of the security module are described in Table 3.3. We use the following new type to describe the return value of a security check:

```
typedef struct {
  key depends[];
  int status;
  value val[];
} check_ret_t;
```

Operation	**Parameters**	**Return value**
`key[] precheck(key k);` precheck estimates additional keys that need to be examined to determine if value corresponding to key `k` is secure for access	`k` – key that should be checked by security module	Array of keys that are required to determine if value corresponding to key `k` is secure for access. This could be empty.
`check_ret_t check(key k[], value v[]);` Checks the set of keys and values for security issue	`k` – array of keys that need to be checked for security issues. This includes the original key and the output of precheck. `v` – array of values corresponding to keys `k`	`.status` – `GOOD` if the set of values `v` has no security issue. Some security modules such as encryption return a new set of values in `val` field `BAD` if the set of values `v` are found to have security issues `INSUF` if more values are needed to decide on security issues in the set of values `v`. The keys whose corresponding values are needed are populated in `depends` field. `.depends` – List of keys whose values are required to assess the given set of keys. This filed is valid only if `status` is set as `INSUF`. `.val` – Array of new values that the security module generated. Whether this field is used really depends on the type of security the module enforces. The field is valid only if `status` is `GOOD`.
`void security_update();` Call back function for any update to security module		
`void report();` Report to administrator in case of security breach. This may involve logging values, usernames, keys, etc.		

Table 3.3: Simple security interface

In case of security, we factor the possibility of security upgrades where the cached results might have to be purged. This is particularly relevant in case of anti-virus, where there could be updates to signature databases on a daily basis. The call back for this is `security_update` as listed in Table 3.3.

Extending the Simple Model to the File System Interface

Some references to a file model is already made in examples in previous sub-sections. Here, we show how the key–value model works with a file system interface.

A key translates to a tuple ⟨*file handle*, *offset*, *length*⟩. Value is the data in file at *file handle* in the range [*offset*, *offset* + *length*).

As for protocol operations, read and write have additional open and close. Thus read translates to open–read–close and write translates to open(O_CREAT)–write–close. Remove can be thought of as punching hole in file. When the entire file is hole, remove the file. With this, read has to be redefined to return `NOT_FOUND` when reading part of a hole.

More formally, we have definition of key and value:

-
  ```
  typedef struct {
    const char *path;
    size_t offset;
    size_t length;
  } key;
  ```

-
  ```
  typedef struct {
    char *data;
    size_t length;
  } value;
  ```

As a simple example, `ret_t read(key k)` translates to following in server storage:

```
ret_t read(key k) {
  ret_t r = {{NULL, 0}, NOT_FOUND};
  int fd;
  ssize_t read_ret;
  if ((fd = open(k.path, O_RDONLY)) == -1) {
    r.err_code = NOT_FOUND;
    return r;
  }

  r.val.data = malloc(k.length);
  r.val.length = k.length;
  // Assumption: read returns all data unless there’s an end-of-file
  read_ret = read(fd, (void*) r.val.data, r.val.length);
  if (read_ret != r.val.length) {
    r.err_code = NOT_FOUND;
    r.val.data = NULL;
```

```
    return r;
  }

  return r;
}
```

3.3.2 Ordering of Security and Cache Layers

In this section, we analyze the effect of ordering security and cache layers. We look into the interaction between the modules for two major operations – read and write.

Approach A: Security Layer Above Cache Layer

In approach A, we consider security layer above cache layer in proxy. Thus, the security module is closer to client. We analyze read first, followed by a write.

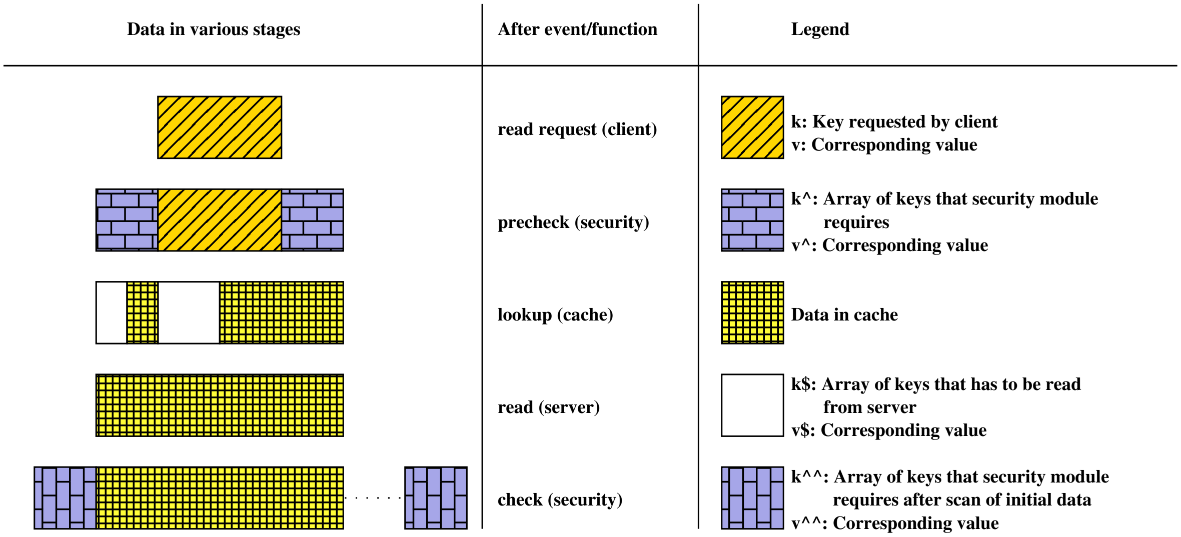


Figure 3.2: Approach A: Data in various stages of read request processing

Read. Read flow is depicted in Figure 3.2 and Figure 3.3. Figure 3.2 shows a pictorial view of how data progresses down the layers whereas Figure 3.3 shows the flowchart diagram. Figure 3.2 also serves as reference for terms used in flowchart.

The client's read request is denoted by key `k`. As mentioned earlier, this corresponds to a file–offset–length. The `precheck` function in case of integrity or encryption will extend the read request to block units. In case of a signature matching anti-virus, the `precheck` might add cushions [29] on either side to have enough bytes to match maximum pattern length. `k^` corresponds to array of keys that correspond to this additional data. The request to cache layer is now `k+k^`. The cache might have part of the data already available and the part that is not available is denoted by `k$`. The request to server is this. Once `k$` is fetched from server and inserted to cache, the values corresponding to `k+k^`, `v+v^` is returned to security layer. Here, it performs a `check` on `v+v^`. In case of anti-virus, this might lead to partial match or

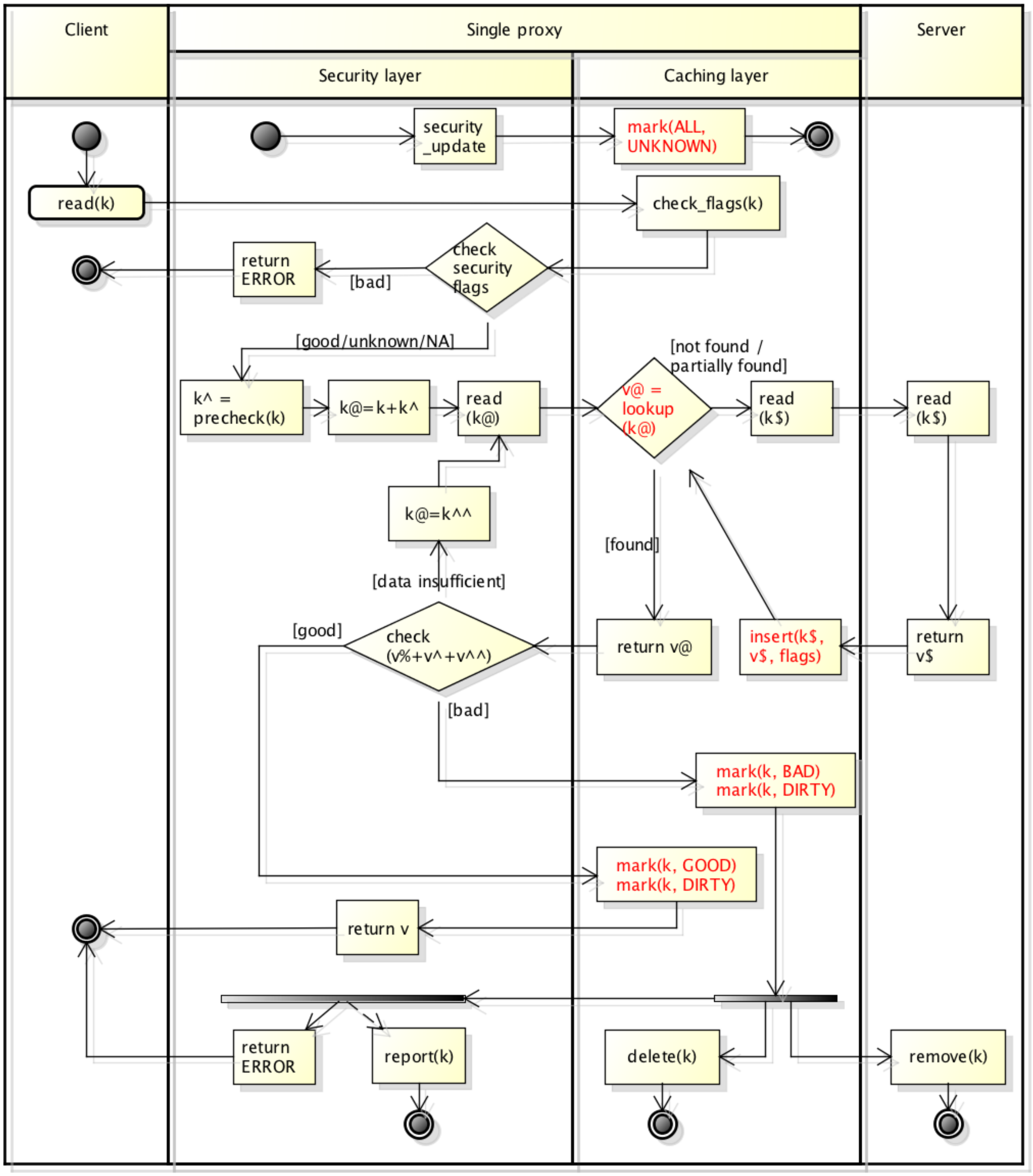


Figure 3.3: Approach A: Flowchart for read

a multi-part pattern and need other parts of the file to be read in order to decide on whether the data is infected or not. While the data identified as needed by precheck is assumed to be immediately after or before the requested data, the extra data required by check could be elsewhere in the same file. This extra data is depicted using k^^ and is read from the underlying cache layer. Once v^^ that corresponds to k^^ arrives, the check function gives verdict on security of the v+v^+v^^ possibly after more iterations of additional data requests k^^. The additional requests (k^^) are not expected in case of integrity check and encryption. Another point to note is that the value itself is modified by check if it is a crypto module. It should basically do a decryption each time data is read by client.

Note that in order to optimize security check, we might cache GOOD/BAD for each key. This helps malware scanning and integrity check to optimize the check function. Decryption does not benefit from this and has to be done for every read. There is a possibility of writing clear text to cache layer, but we have to make sure that only encrypted text gets written back to server. In our design, we do not let the cache maintain both clear-text and cipher-text versions of the same data. Hence, we cannot expect good performance for decryption with approach A.

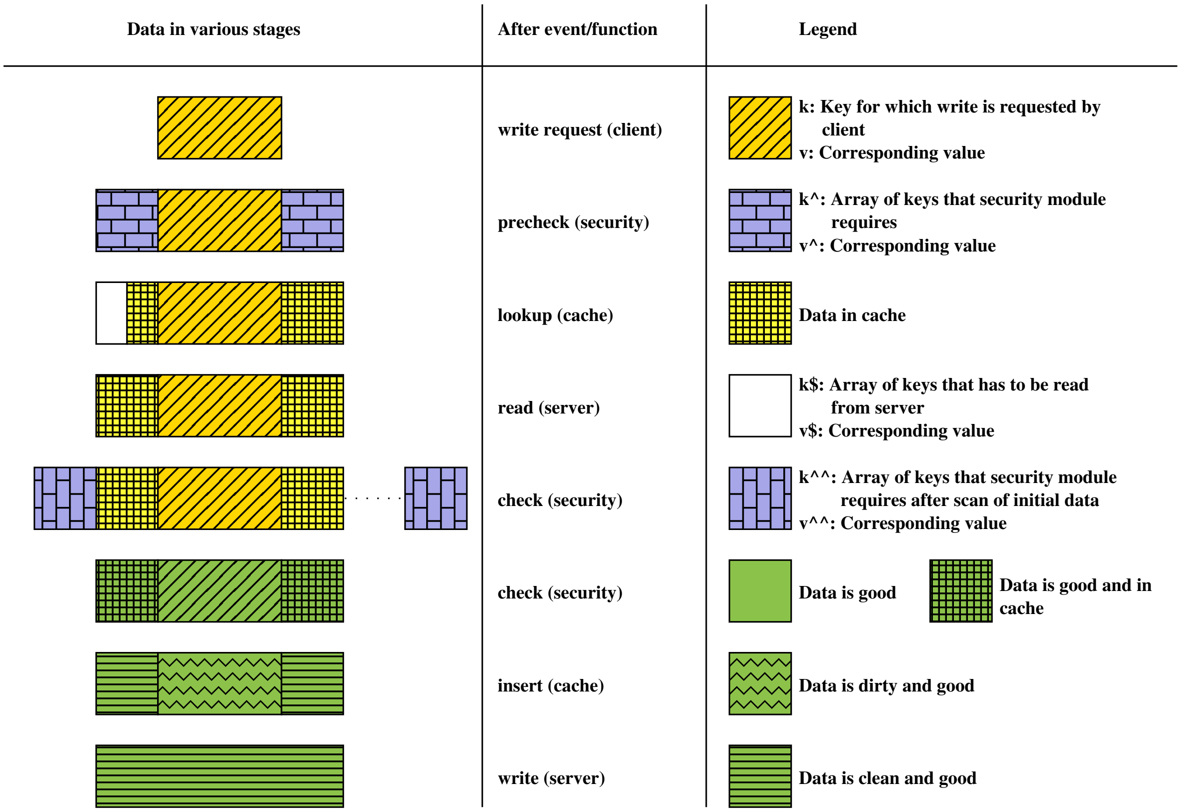


Figure 3.4: Approach A: Data in various stages of write request processing

Write. The write flow is depicted in Figure 3.2 and Figure 3.3. Figure 3.2 shows a pictorial view of how data progresses down the layers whereas Figure 3.3 shows the flowchart diagram. Figure 3.2 also serves as reference for terms used in the flowchart.

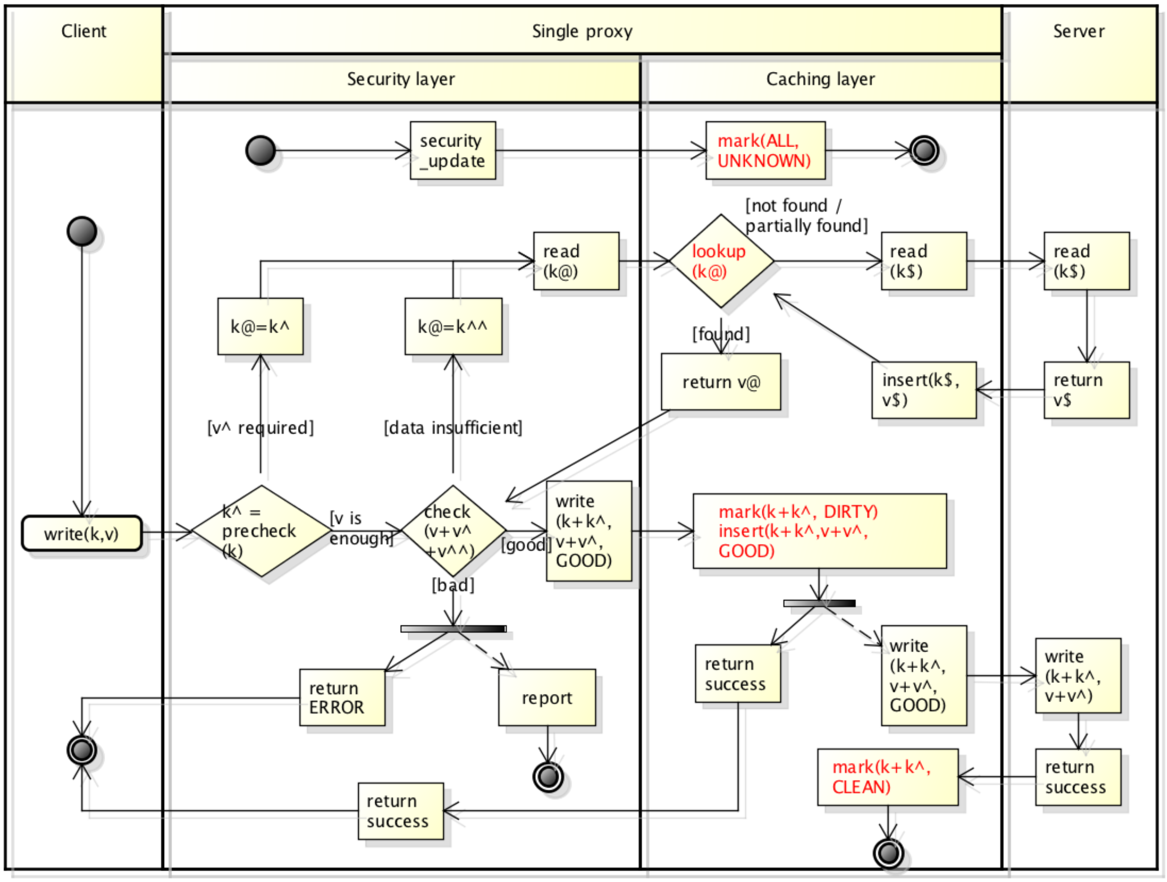


Figure 3.5: Approach A: Flowchart for write

A client's write request is denoted by key `k`. Similar to read request flow, `precheck` estimates the data required by security modules. As discussed earlier, the estimated data may be transforming to block units in case of integrity and encryption and cushion data in case of anti-virus. The estimated data request is denoted using `k^`. Once this data is read from cache, corresponding values `v+v^` are checked for security issues. If the security module does not arrive at a conclusion based on available data, it might need further data, denoted by `k^^`. After multiple iterations, the security module will scan or process `v+v^+v^^` and determine whether the data is good to go. If not, this module returns an error to client and reports the event to administrator. Once a clearance is obtained, the data is inserted to cache and marked as `GOOD` and `DIRTY`. Client is acknowledged and at some stage, based on cache mechanism and policy, a write back to server is initiated. Once server write back finishes, the key is marked as `CLEAN`.

In the flowchart, we have shown the write to cache to include `k+k^, v+v^`. This is not necessary for anti-virus, as only the new data `k, v` need to be inserted to cache. In case of encryption, we need to write in block units and hence `k+k^, v+v^` is required. Even in case of integrity, the checksum has to correspond to the entire block, though actual data write may be only the new data `k, v`. Though not indicated in diagram, the meta-data may not indicate just a flag such as `GOOD/BAD` but also have additional information such as checksum.

Note-worthy points. One interesting point to note here is that a write-back cache is perfectly alright in approach A. Even with a write-back cache, the client gets to know any rejection of read or write request synchronously. There is one catch, though. If the security module gets updated, then there could be a case where client is acknowledged with a success, whereas a security upgrade causes the data to be quarantined or removed without the client's knowledge.

Another positive with approach A is that all data, even the extra ones requested by security module, are cached. This reduces server accesses significantly and thus gives better performance. The flip side is that all encryption and decryption have to be done for all access requests.

Approach B: Cache Layer Above Security Layer

In approach B, we consider cache layer above security layer in proxy. Thus, security module is closer to server. We analyze read first, followed by write.

Read. Read flow is depicted in Figure 3.6 and Figure 3.7. Figure 3.6 shows a pictorial view of how data progresses down the layers whereas Figure 3.7 shows the flowchart diagram. The Figure 3.6 does not cover all cases in flowchart, but serves as a reference for terms used in flowchart. It also does not show the insert back to cache after security scan.

Write. Write flow is depicted in Figure 3.8 and Figure 3.9. Figure 3.8 shows a pictorial view of how data progresses down the layers whereas Figure 3.9 shows the flowchart diagram. The Figure 3.8 does not cover all cases in flowchart, but serves as a reference for terms used in flowchart. It also does not show the insert back to cache after security scan.

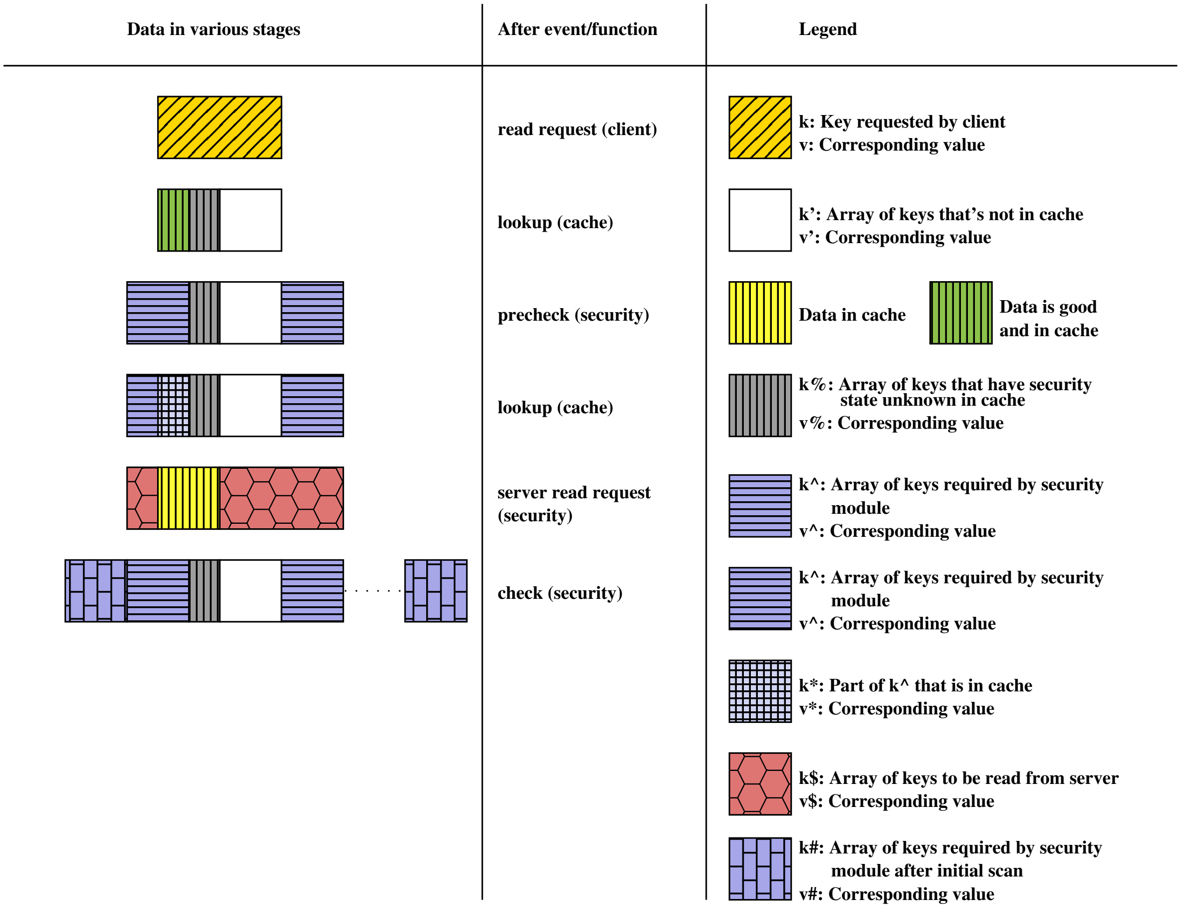

Data in various stages
After event/function
Legend
read request (client)
lookup (cache)
precheck (security)
lookup (cache)
server read request (security)
check (security)
k: Key requested by client
v: Corresponding value
k': Array of keys that's not in cache
v': Corresponding value
Data in cache
Data is good and in cache
k%: Array of keys that have security state unknown in cache
v%: Corresponding value
k^: Array of keys required by security module
v^: Corresponding value
k^: Array of keys required by security module
v^: Corresponding value
k*: Part of k^ that is in cache
v*: Corresponding value
k$: Array of keys to be read from server
v$: Corresponding value
k#: Array of keys required by security module after initial scan
v#: Corresponding value

Figure 3.6: Approach B: Data in various stages of read request processing

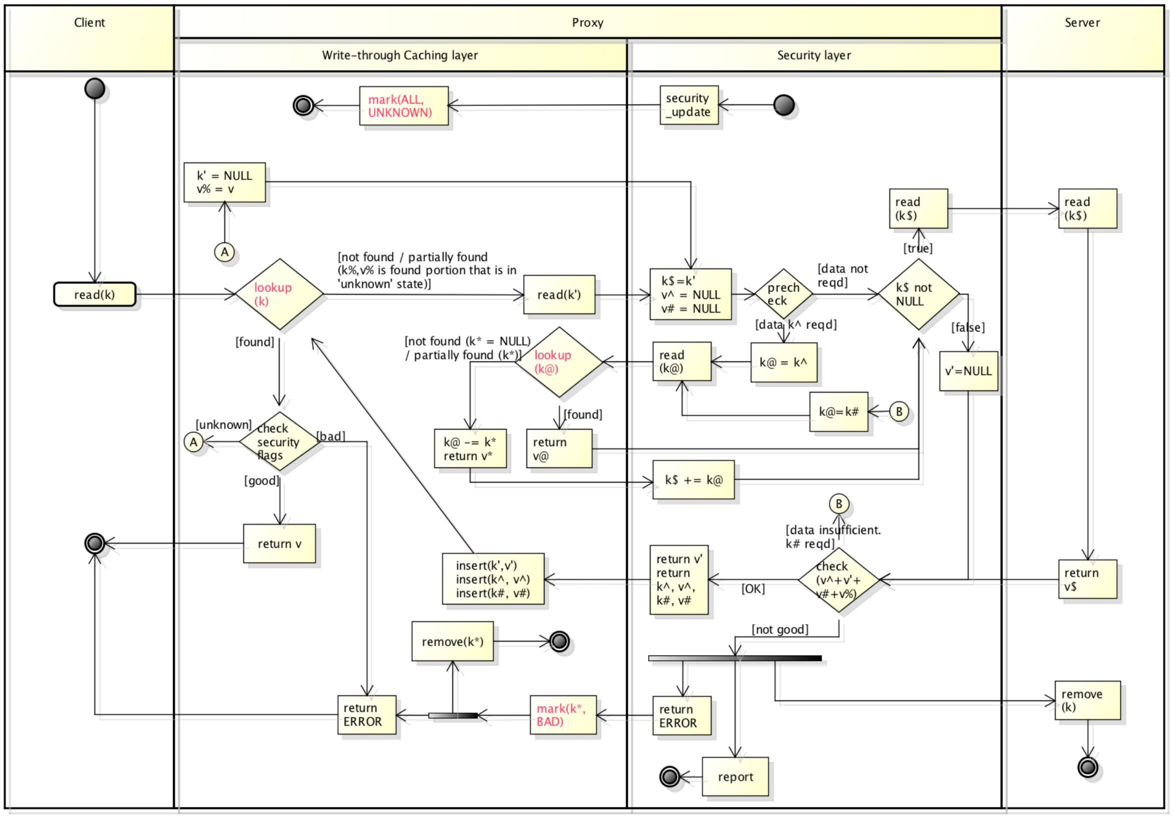


Figure 3.7: Approach B: Flowchart for read

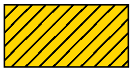
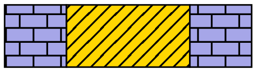
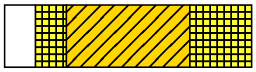
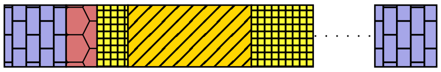
Data in various stages	After event/function	Legend
	write request (client)	🟨 k: Key for which write is requested by client v: Corresponding value
	precheck (security)	🟦 k^: Array of keys that security module requires v^: Corresponding value
	lookup (cache)	▦ k*: Part of k^ that is in cache v*: Corresponding value
	check (security)	□ k$: Array of keys that has to be read from server v$: Corresponding value
		▥ k^^: Array of keys that security module requires after scan of initial data v^^: Corresponding value
		🟥 Data fetched from server

Figure 3.8: Approach B: Data in various stages of write request processing

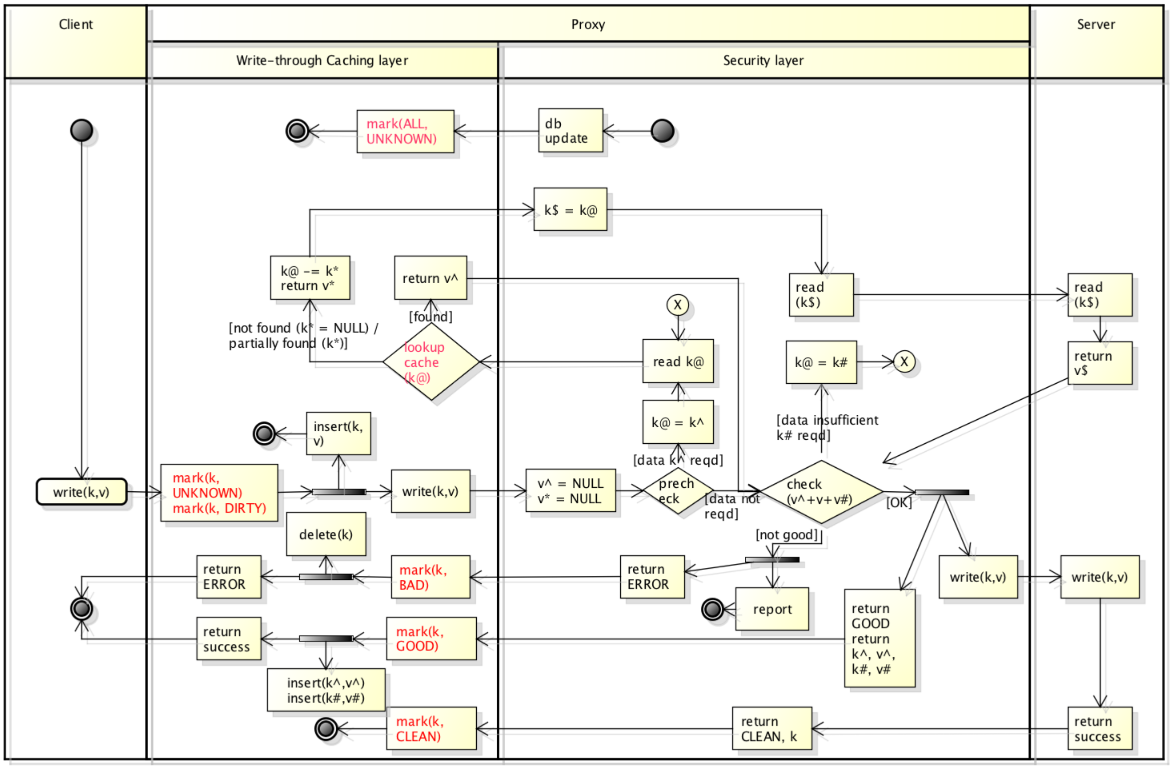


Figure 3.9: Approach B: Flowchart for write

Note-worthy points. The main issue that can be observed with approach B is that there are a lot of violations of the layered architecture. This can, of course, be avoided by not letting the security module to access the cached contents and serving all read requests from server. The fixing of violation will result in sub-optimal performance since server accesses have to go over the WAN. A security update will also result in violation where cache module has to be accessed by security module to invalidate status in cache.

The main reason for violation can be attributed to additional data required by security module. This cannot be avoided for anti-virus scanning since there are all these complex patterns. However, we can pre-determine data requirements for encryption and integrity. By decrypting when data is read from server and encrypting when data is written to server, the cache always contains cleartext. This avoids performance delays that would come with approach A.

3.3.3 Coming Up with the Final Design

The analysis of previous sections have helped us arrive at some interesting trade-offs with ordering of cache and security layers one way or the other. This is summarized in Table 3.4.

The security modules that we plan to have in our system are anti-virus, integrity, and encryption. Of these, integrity check is for ensuring data-integrity in server; encryption is done to protect data confidentiality in server; and anti-virus is to protect system from malware that will usually come from clients. The requirements of each security module needs to be

Item	**Approach A** **Security layer above cache layer**	**Approach B** **Cache layer above security layer**
Write-back cache	Write-back cache can be supported easily. Dirty data in cache is always good.	Write-back cache cannot be supported if client has to get success/failure response from proxy. The success of a write cannot be guaranteed at cache layer.
Bad data in cache	Bad data can come from the server side	Bad data can come from the client side
Additional data required by security	Since cache is below security module, all extra data required by security module can be read from cache	Since security module is below cache, extra data required by security module leads to violation of layering principles
Caching the additional data required by security	Since cache is near to server, all data read from server is cached	Since cache is near to client, cache needs to accept more than the data it requests from security module. Thus, there's violation of layering principles.
Modification of data by security	Cache contains modified (cipher-text) data. All data accesses involve computation by security module. Though it is possible to maintain both cipher-text and clear-text in cache, there is either an overhead of space or meta-data management.	All computation involved with modification (such as encrypt/decrypt) can work below cache. This enables cache to maintain clear-text all the time and let security module worry about translations at the time or read from or write back to server.
Handling security updates	Invalidate cache and check security versioning in cache-metadata	Invalidate cache and checking security version in cache-metadata at the expense of violation of layered architecture principles

Table 3.4: Comparison of different layer orders of cache and security

understood in order to find out how to fit them around cache in layers.

The requirements of anti-virus module are as follows:

- Bad data from client should be guarded against. Hence all client writes should go through the scanner. If the system is protected by integrity, then data from server can be assumed to be safe to access. We can do away with scanning for data reads.
- Anti-virus has complex data requirements to check security. In order to determine if a piece of data is good or bad, it would need several other bytes from the file. These data requirements can often not be determined in advance and might happen as the scan progresses.
- When there's a malware signature database update, we should make sure that all data is scanned against new set of signatures. One way to do this is to scan every time. Another way is to maintain the database version in file or cache metadata. If this information is kept in server (and system has integrity checks) then scans can be optimized to once per file per signature. If we keep the information in cache, then we can scan every time the data is inserted into cache after a previous eviction.

Requirements of anti-virus module suggests putting it closer to client and above cache layer. This also enables using a write-back cache, which is more optimal.

The requirements of integrity module are as follows:

- Integrity of data from server should be protected. Hence all reads from server should go through integrity check. For writes to server, a new checksum has to be calculated before write. With these requirements, integrity module is best close to server since, in this way, all data in cache is assured to be valid.
- Integrity does not have a success or failure for clients write. Hence, approach B does not is not unsuitable, as mentioned in Table 3.4.
- Though integrity module would need additional data for reads and writes, it is predetermined. This means we can keep approach B, optimally and without layering violations, if we make sure all accesses are aligned to block units by the cache.

Requirements of integrity module show that it will work well if it is closer to server, with the little tweak of cache accessing integrity layer in block units.

The requirements of encryption module are as follows:

- Data read from server should be decrypted and data written back to server should be encrypted. Since it is protecting data confidentiality from server, the layer below cache is suitable for crypto module. This way, cache can always contain clear text.
- Crypto module does not have a success or failure for clients write. Hence, approach B is not unsuitable, as mentioned in Table 3.4.
- The data requirements of crypto module is similar to integrity, as in, it is in block units. We assume for simplicity that both crypto and integrity module work with the same block-size. Hence apporach B can be used for crypto with the same block-unit-access requirement as in integrity.

Requirements of encryption module, or more aptly, the crypto module indicate that approach B fits. Block-unit-access from cache ensures caching of all data accessed from server without violating stach architecture principles.

With the above in mind, the final architecture—a hybrid of approaches A and B—is presented in Figure 3.10. Flowchart diagrams for read and write are captured in Figure 3.11 and Figure 3.12, respectively.

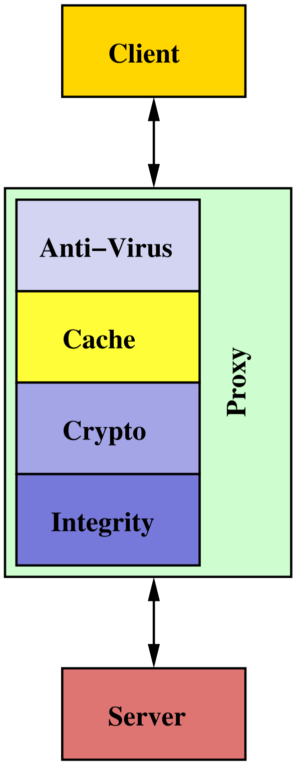


Figure 3.10: Stacking security and cache modules in proxy

It is interesting to note that in the final design in Figure 3.10, the cache is sandwiched by security modules as if to fend off security threats from respective sides. That is while anti-virus protects cache against malware from clients, the integrity-check and encryption modules protect data integrity and confidentiality of data on the server.

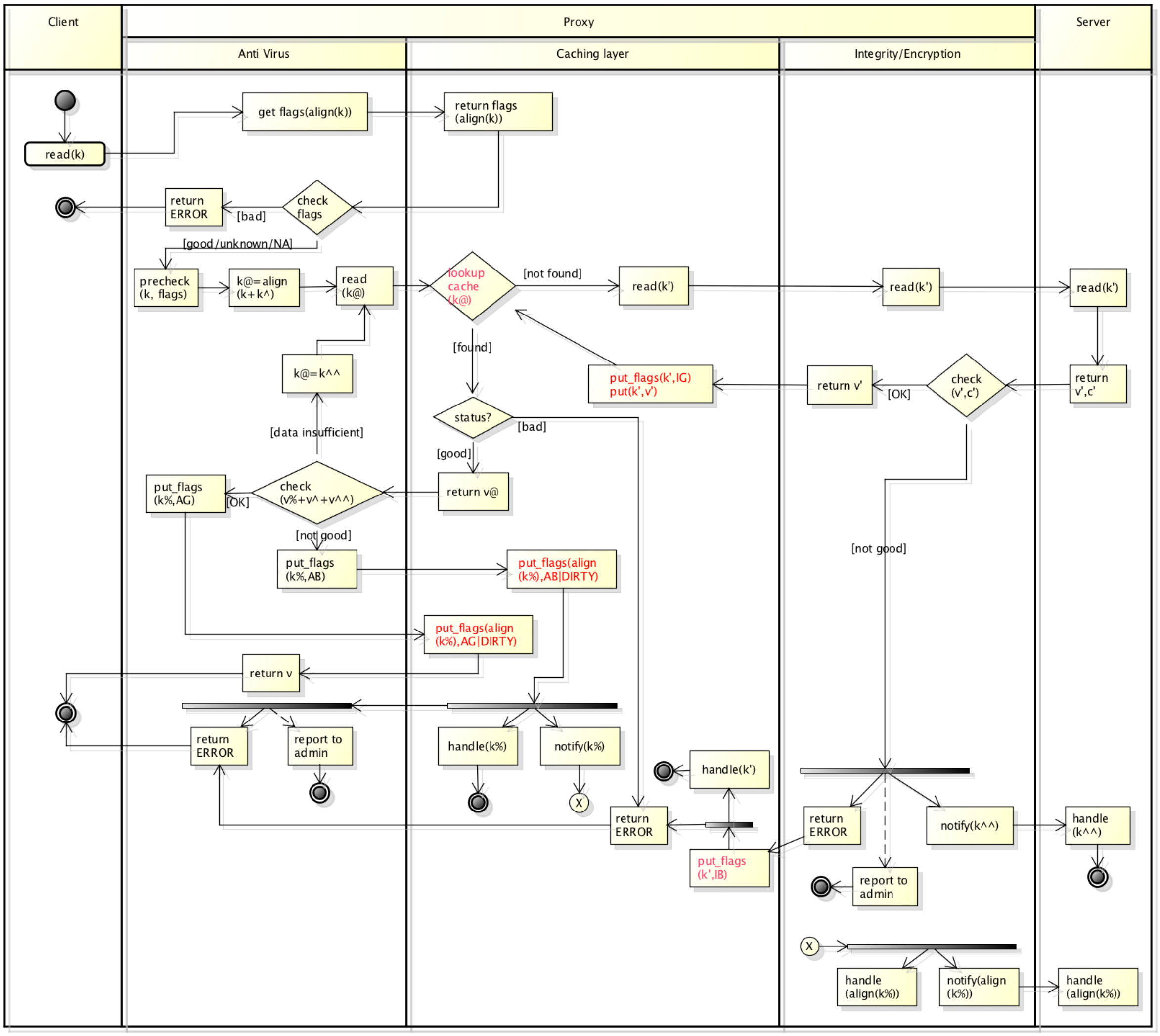


Figure 3.11: Flowchart for read with anti-virus, cache, crypto, and integrity modules

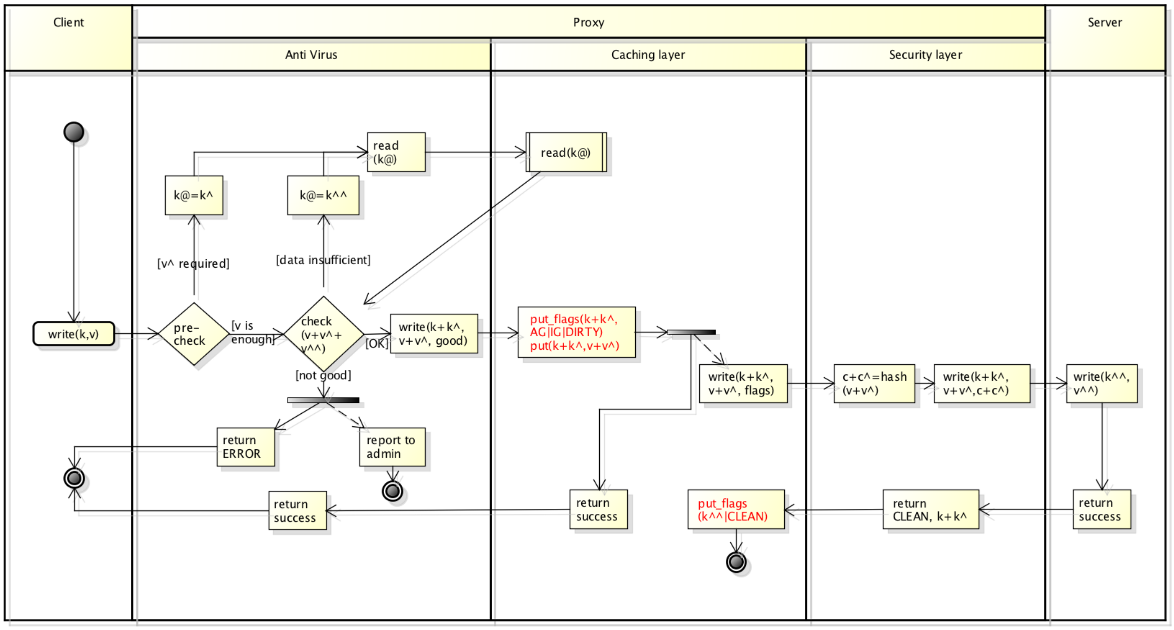


Figure 3.12: Flowchart for write with anti-virus, cache, crypto, and integrity modules

Chapter 4

Implementation

In this section, we give details of implementation of the system. We start with information about the basic details and then talk about cache, anti-virus, integrity and encryption modules.

4.1 Network Storage Architecture with Proxy

Our system has a set of clients, a proxy machine and a server machine for data storage. The standard used for data exchange is Network File System (NFS). NFS is widely used for networked storage. Version 4 of NFS is designed to work over the Internet (WAN) [7].

As established earlier, all our security enforcements are in the proxy machine. The proxy should be able to act as NFS server to clients and as NFS client to storage server. It should also support implementation of cache and security features in a layered manner. To implement this, we used a user-land implementation of NFS server, the NFS-Ganesha.

4.2 Anti-Virus

We use ClamAV [24] as our anti-virus engine. ClamAV can be linked as statically with an FSAL shared object. Malware databases are downloaded manually, using an executable "freshclam" that comes with ClamAV source.

Once initialized with a malware signature database, the vanilla ClamAV accepted a file path or a file descriptor as input and scans the entire file against viruses. We modified ClamAV and added a scaning function to accept a memory buffer, which contains the data of a whole file. This is reasonable because viruses tend to infect small files, and it is a common practice to scan only small files. For example, the popular email service, GMail, only scan attachments smaller than 10MB. In our implementation, the size threshold of anti-virus scanning is a configurable parameter with a default value of 10MB.

Miretskiy et al. optimized ClamAV and used the engine to incrementally scan data in an on-access anti-virus stackable file system called Avfs [29]. The type of signatures at that point were either a regular pattern or a multi-part pattern. These were matched using a variation of Aho-Corasick pattern-matching algorithm [2]. Signature types in ClamAV [11] have evolved considerably over the last decade and most signatures are based on hashes of either the entire file or part of file such as section. The original pattern based signatures are less than 1% of the entire database [24].

The anti-virus FSAL implementation does not maintain version of malware database that was used to scan a file. Once a file is scanned, it is assumed to be free of virus. Although not ideal, this implementation is enough for this study as we are focusing on the trade-off between security and performance.

We use a full-file buffer scanner using ClamAV with an anti-virus FSAL module. Since integrity FSAL is present, the server could not infect files with viruses without being caught. Therefore, the anti-virus FSAL scans files only on write. Ideally, client that requests the write should know whether the write succeeded or not with a return value. In our implementation, clients do not get to know from return value whether the data that is written contains malware. We decided to trade-off this limitation for a better performance by scanning the changed content asynchronously at a configurable interval. We designed the anti-virus FSAL to scan a file that is open for read-write:

- every configurable interval after the time of open (5 minutes by default)
- at close of file

The overhead of scanning the entire file for every write would be huge if the file is large, say, a virtual disk image file.

For each file that a client opened for read-write, the anti-virus FSAL maintains a pointer to `fsal_obj_handle`, and the last scan-time in a node. For files that are not scanned yet, the node keeps time of open, instead. For fast ($O(\ln n)$) lookup, the anti-virus FSAL maintains the nodes in an AVL-tree with address of `fsal_obj_handle` as key. The anti-virus FSAL also keeps the nodes in a queue (implemented as a doubly-linked-list), in increasing order of scan-times. The anti-virus FSAL uses a timer thread to check if there are any files whose scanning is due. This thread wakes up every 5 seconds and collects nodes from the head of the queue. All nodes at the head of queue that are past their scan-times by more than the configured interval are dequeued and passed to a newly spawned scanner thread. The scanner thread scans the files corresponding to nodes. Scanning of a file is done only if it has dirty data. Otherwise, the current time is noted as scan-time and no scanning is done. Once scanning of a particular file starts, the anti-virus FSAL does not permit any concurrent writes to that file. The FSAL allows further writes to the file only after the scanned contents are written down to next layer. This ensures that only scanned contents are written back to server. Once scanning is over, the nodes are enqueued again. A node keeps this dequeue and enqueue process util it is reclaimed when the corresponding file is closed.

In the current implementation, when a file is found to be infected, there is no way of informing the client that an earlier write was unsuccessful. This is because, anti-virus FSAL performs scan asynchronously, as mentioned earlier. In our implementation, the proxy-cache FSAL invalidates the cache. Subsequent reads would hence result in reading of good data from server. Thus, if a file is infected, an earlier version of file is retrieved. The flip side of dealing with malware detection in this way is that client is not informed. Other way of dealing with detection of malware in files is to return error, such as, NFS4ERR_IO on subsequent open by any client. This is not implemented in anti-virus FSAL yet.

4.3 Proxy-Cache FSAL

The proxy-cache FSAL is responsible for maintaining a data-cache in proxy machine. The data-cache has to support persistent storage, since the COMMIT operation of NFSv4 requires that data is flushed to stable storage [36]. Also, since data has to eventually reach the server, states of cached content such as "dirty" should also be stored in disk. This will ensure effective crash-recovery.

The proxy-cache FSAL is designed to be a write-back cache, to minimize server accesses. This way multiple overlapping write requests over a duration can be merged and then sent to server together. This also minimizes the overhead of overwrites. The cache write-back is designed to perform a write-back for all files that are open for read-write:

- every configurable interval after the time of open, such as 5 min
- at close of file

The resemblance of write-back policy of proxy-cache with scan policy of anti-virus is no coincidence. In fact, we designed anti-virus to be part of the proxy-cache FSAL. That is, there is no separate FSAL that handles anti-virus alone. Thus every few seconds after a file is open for read-write, there is a scan-and-write-back operation that first performs an anti-virus scan and then a write-back to below layers. The same scan-and-write-back is performed even at close of file. At all other times, writes translate to "insert" into cache and does not go down to further layers. The details of timer-thread, scanner-thread, and data-structures involved are already captured in Section 4.2.

Reads to proxy-cache FSAL works as with a regular cache. Read to proxy-cache looks up for data in cache module first. If data is not available, proxy-cache FSAL passes down request to underlying layers and inserts the replied data to cache once the request is done.

Cached contents and some meta-data related to the cached contents are maintained on local storage of proxy machine. Meta-data about cached contents adds to cache-recoverability after a crash. This is especially important when there are dirty data in cache that has not been flushed to server. Storing cached contents stably is also required since NFSv4 server must have "stable storage" [37].

For every file in the back-end server that is cached, the proxy-cache FSAL maintains a sparse file in local storage. For the portion of data that is cached, the FSAL writes to corresponding offset in the local file. Removal of extents are done by punching holes to file. The presence of holes ensure that local storage does not run out of disk space too fast with sparse files. This design simplifies the caching by delegating file block management to the proxy's local file system. Note that, proxy-cache can also implicitly keep the cached data in the proxy's memory through the OS's page cache.

We designed the proxy-cache FSAL, carefully, to be thread-safe. We use file-specific range-locks to make sure reads and writes to overlapping locations in cache does not happen. We also handle the asynchronous write-back carefully so that there are no race conditions between write-back threads and inserting threads.

4.4 Integrity and Encryption

Integrity check and data encryption are handled in a separate FSAL layer (module) at proxy-side. On write operations, the module encrypts the plain text, computes the Message Authentication Code (MAC) and stores the cipher text and MAC at remote storage server. On read operations, the module decrypts the cipher text and verifies the MAC. A file is divided into multiple blocks and authenticated encryption is applied on a per-block basis. In the following paragraphs, we describe the security algorithm, key management, data organization and data path.

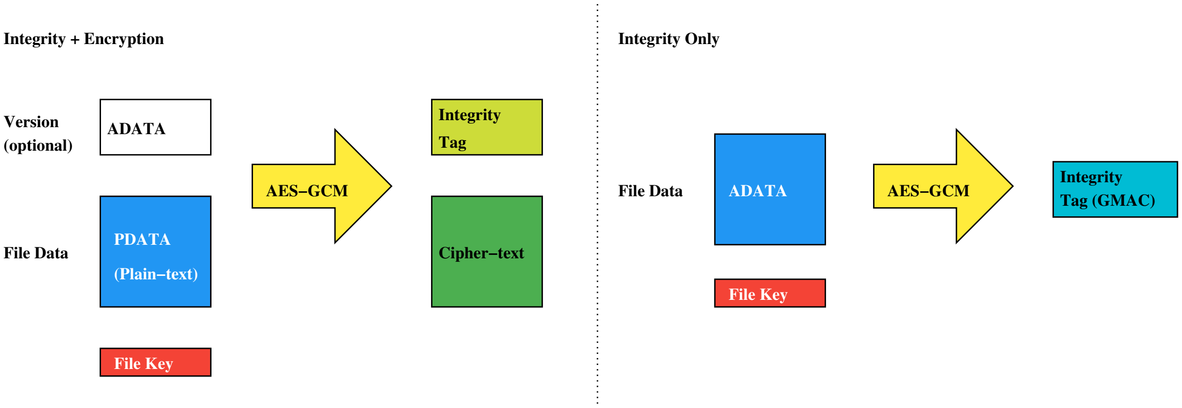


Figure 4.1: AES-GCM for integrity and optionally encryption

We use Advanced Encryption Standard (AES) as symmetric key cryptographic block cipher and Galois/Counter Mode (GCM) as mode of operation to provide authenticated encryption. GCM accepts two input channels: PDATA that receives both confidentiality and authentication, ADATA that is additional authenticated data. To enable both integrity check and encryption, we put data block into PDATA, producing cipher text and tag (i.e, MAC). To protect only integrity, data block is put into ADATA and the resulting integrity tag is also called Galois Message Authentication Code (GMAC), illustrated in Figure 4.1. Potentially, we can authenticate data block with its version number to defend against replay attack, which requires a proxy-side database to keep block version numbers.

Key management is critical in encryption context. On file creation, we generate a cipher key exclusively for that file and then encrypt the generated file key with proxy's own public key [20] with RSA algorithm. Encrypted file key, together with other encrypted attributes (e.g., file hole record), is stored in the first (one or two) block of the file as file header, depicted in Figure 4.2. To make a file accessible to a new proxy, existing proxy encrypts the file key with new proxy's public key and adds an identifiable entry in header. This approach suggests that the proxy only needs to maintain its own public/private key pair and other proxies' public keys. On file lookup operation, the first thing for a proxy is to find its entry in header and retrieve the file key with its private key. On file close, the proxy writes back the header if proxy entries or attributes are modified. The module will cache the encrypted file key portion of header instead of performing encryption each time on write-back because it is rarely updated. Though it is expandable to multi-proxy environment, we focus on single proxy setup at this stage.

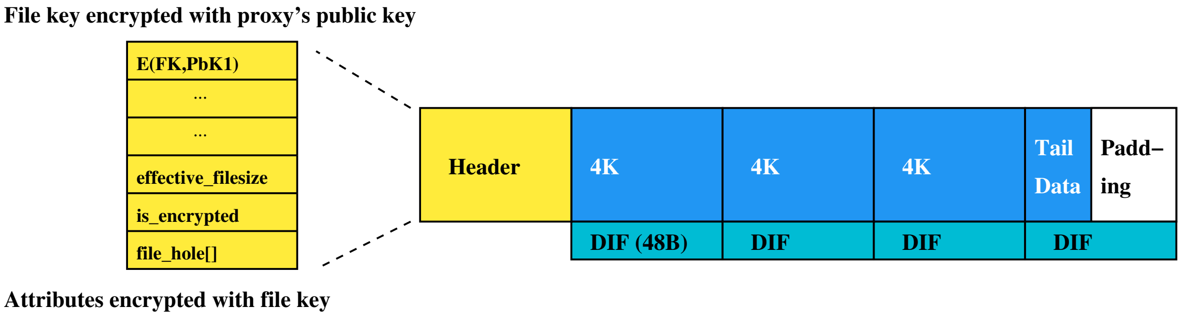


Figure 4.2: Header, data blocks and data integrity field of a typical file

We leverage Data Integrity eXtensions (DIX) [14] to store integrity tags (i.e, MAC) at server-side instead of proxy-side, which eliminates block-to-tag bookkeeping. DIX extend the disk sector from traditional 512 bytes to 520 bytes by adding 8 protection bytes, out of which at most 6 bytes are available to applications. We aggregate several sectors to be one logical block so that we have sufficient space for 16 bytes tag and other potential meta-data. Specifically, we use 4096 bytes as our block size which is made up of 8 sectors, and therefore we have in total 48 bytes Data Integrity Field (DIF) on disk for each block, depicted in Figure 4.2. When storage server does not opt in DIX, proxy's local database can be used to maintain tags with careful mapping.

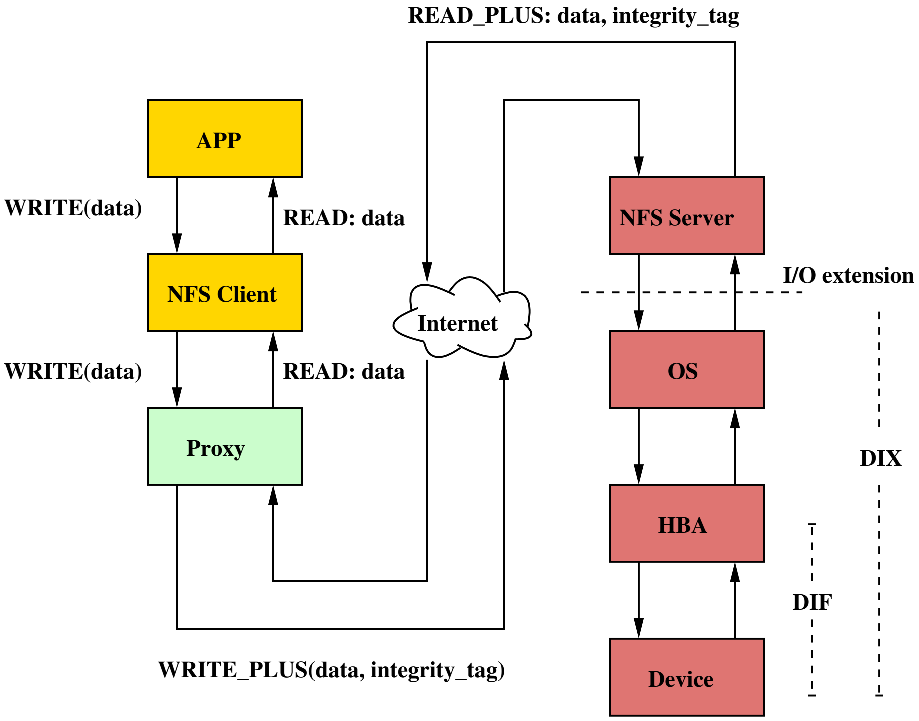


Figure 4.3: NFS End-to-end Data Integrity

To convey tags from proxy to server on read and write, we implemented NFS End-to-end Data Integrity extension [33] in NFS-Ganesha. The extension specifies the protocol to pass protection data with NFS READ_PLUS and WRITE_PLUS operations. In Figure 4.3, the paths in bold are where the NFS extension is applied. At server-side, we use Linux I/O extension [35] to commit data blocks along with their associated tags to the storage device synchronously.

4.5 Proxy FSAL

Proxy FSAL is the underlying, non-stackable, "native" FSAL in the proxy machine. This is the layer that talks to server. The proxy FSAL is already part of NFS-Ganesha. However, we have improving the proxy FSAL in many aspects including fixing bugs, resolving race conditions, and optimizing performance. We have contributed our improvements back to the open-source community.

Proxy can be configured with server address and mount point in server. There are also other configurations such as the maximum read and write size for NFS operations to server.

4.6 Development Effort

Table 4.1 captures code size of main components that were developed as part of our research. In addition to this, we added features and fixed bugs in NFS-Ganesha and ClamAV.

Module	**Language**	**files**	**blank**	**comment**	**code**
proxy-cache and anti-virus FSAL	C	8	272	414	1649
	C++	38	909	470	4949
	C/C++ Header	44	768	1349	2504
	CMake	8	41	11	162
	SUM:	**98**	**1990**	**2244**	**9264**
integrity and crypto FSAL	C	6	374	427	1560
	C++	10	469	119	1486
	C/C++ Header	6	147	225	472
	CMake	1	23	4	68
	SUM:	**23**	**1013**	**775**	**3586**
TOTAL:		**121**	**3003**	**3019**	**12,850**

Table 4.1: Development effort in project

Chapter 5

Evaluation

This chapter presents the evaluation of the secure proxy. Before running experiments, we have verified the correctness of our implementation using `xfstests` [46], which is a popular file system test tool. Our secure proxy has passed all applicable `xfstests` test cases when the stackable layers are enabled individually and combined together.

5.1 Experimental Setup

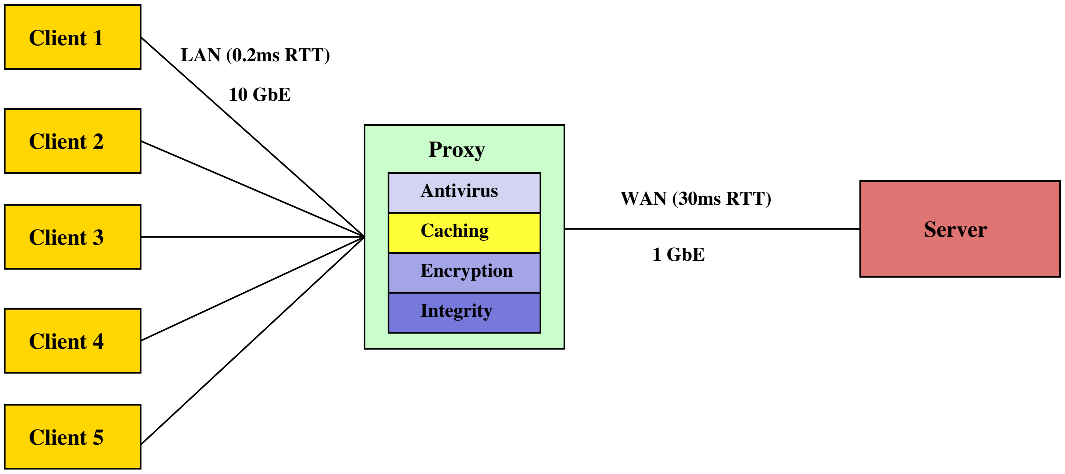


Figure 5.1: Benchmark setup.

Figure 5.1 shows our experimental setup, which consists of seven identical Dell PowerEdge R710 machines. Each machine has a six-core Intel Xeon X5650 2.66GHz CPU, 64GB of RAM, a Broadcom BCM5709 1GbE NIC, and an Intel 82599EB 10GbE NIC. Five machines run as NFS clients, one as the secure NFS proxy, and one as an NFS server. Clients communicate to the proxy using the 10GbE NIC via a Dell PowerConnect 8024F 24-port 10GbE switch; the proxy communicates to the server using the 1GbE NIC via a PowerConnect J-EX4200 48-port 1GbE switch. We measured the RTT between the clients and the proxy to be 0.2ms using `ping`. To emulate the proxy's connection to the in-cloud server via

WAN, we injected a 30ms delay in the outbound link of the server using `netem`. We set the delay to 30ms according to Panache's report of inter-datacenter latency [26].

All machines ran CentOS, a freely available version of Red Hat Enterprise Linux widely used in enterprise environments. The CentOS version is 7.0, and the kernel version is Linux 3.14. To use the Data Integrity eXtension (DIX), we patched the server's kernel with DIX support [35] which allows integrity payloads to be passed from user-land to the kernel. Because we do not have physical storage devices that support DIX yet, we set up a 10GB DIX-capable virtual SCSI block device using `targetcli` [32]. The DIX-capable device is backed by the server's RAM and is formatted with `ext4`. Our DIX capable NFS-Ganesha server exports the device as an NFS file system.

Configs	Proxy	Integrity	Encryption	Caching	Antivirus
P	✔	✘	✘	✘	✘
I	✔	✔	✘	✘	✘
IE	✔	✔	✔	✘	✘
IC	✔	✔	✘	✔	✘
ICE	✔	✔	✔	✔	✘
ICEA	✔	✔	✔	✔	✔

Table 5.1: Combinations of security features.

The proxy also runs NFS-Ganesha, which acts as a client to the NFS server and re-exports the server's NFS file system to the clients. Note that the NFS protocol is NFSv4.2 with the DIX extension [33], whereas the communication protocol between clients and the proxy is NFSv4.0. The difference of the protocols is because the proxy adds additional integrity payload to protect clients' data when the data are written to the in-cloud server. The secure features of the proxy, which are implemented as stackable NFS-Ganesha layers, can be individually turned on or off using a configuration file. Specifically, we benchmarked six different combinations of these features as listed in Table 5.1. To simplify our analysis of caching effect, we used a 16GB `ramdisk` as caching device so that all data could be cached without any cache evictions. We also emptied the cache before each experiment so that we could observe the system's behavior during the whole process when an initial empty cache is gradually filled to full.

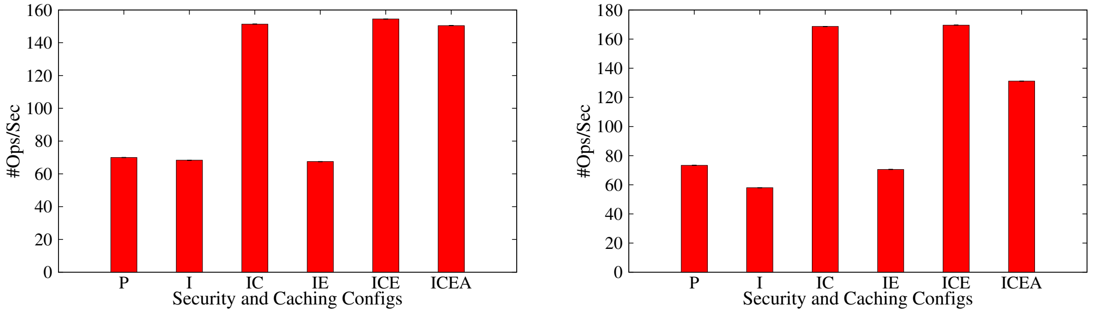
We used *Benchmaster* [7] to perform experiments so that workloads on multiple NFS clients run concurrently. While experiments are running, Benchmaster periodically collects system statistics using tools such as `iostat` and `vmstat`, and by reading `procfs` entries such as `/proc/self/mountstats`. These statistics help us monitor system behavior during the whole life-cycle of experiments. When experiments are finished, Benchmaster also gathers all statistics from clients, the proxy, and the server to a central place for post-analysis.

We benchmarked two sets of workloads: (1) a set of synthetic micro-workloads which pre-configured read-write ratios, and (2) a set of Filebench macro-workloads including File Server, Mail Server, and Web Server. The micro-workloads help us understand the behavior of our secure proxy in a controlled environment; the macro-workloads reflect our secure proxy's performance impact in popular and realistic scenarios.

5.2 Performance of different read-write ratios

This section evaluates our secure proxy's performance impact on workloads with different read-write ratios. Our secure proxy has four features and each of them have different performance impact: caching generally helps performance, whereas integrity, encryption, and anti-virus hurt performance. The performance impact of these features heavily depend on workload characteristics. The read-write ratio is an important characteristic. For example, the performance of a read-only workload is not influenced by anti-virus and benefit a lot from caching. Conversely, a write-heavy workload does not benefit much from caching, and also incurs frequent anti-virus scanning and thus has large performance overhead.

We studied the performance impact of read-write ratio by running workloads with pre-configured read-write ratios. Specifically, we pre-allocated 100 files, and then repeated the following operation for two minutes: randomly pick one file, open it, perform n 4KB reads and m writes at random offsets, and close it. We varied n and m to control the read-write ratio, and compared the performance of the configurations shown in Table 5.1. We tried two file sizes: 1MB and 10MB.



(a) 1M Files

(b) 10M Files

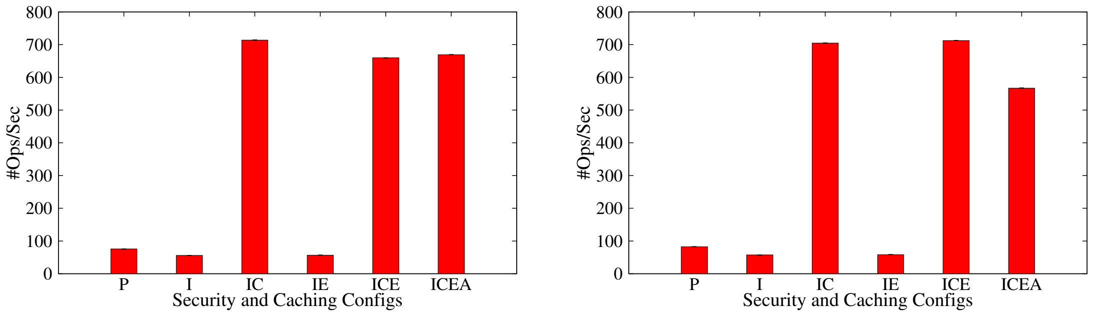
Figure 5.2: Throughput of 1:1 read-write ratio.

Figure 5.2 shows the results when $n = 1$ and $m = 1$, i.e., with a read-write ratio of 1. Overall, the configurations with caching (i.e., IC and ICE) significantly outperformed their counterparts without caching (i.e., I and IE) by 2–3×. Compared with the baseline configuration with only proxy (P), integrity (I) has 3–26% lower throughput because integrity performed more computation to calculate the message authentication code (MAC). Integrity also performed more I/O operations to read and write the file headers that contain file keys and effective file sizes. These I/O operations are slow because of the WAN latency between the proxy and the server. The configuration of I and IE performed around the same despite that IE has to decrypt (encrypt) for each read (write) operation. This is because the overhead of extra CPU computation is negligible compared to the overhead of extra I/Os over the WAN.

As shown clearly in Figure 5.2, adding caching to integrity (i.e., IC) greatly improves the performance and makes the throughput more than 2× higher than the baseline (P). When encryption is further added, the performance of ICE does not drop compared to IC because the caching layer is stacked above the encryption layer and stores data in plaintext form. Further adding anti-virus to ICE only cause a performance penalty of 3% for 1MB files, and of 23% for 10MB files. This is as expected because the anti-virus engine performs full file scanning, and the overhead is higher for large files.

In summary, for this workload with read-write ratio of 1:1, the configuration with all features (i.e., ICEA) performed 2.1× and 1.8× than the baseline (i.e., P) for 1MB and 10MB files, respectively.

The caching layer in our secure proxy is a write-back cache backed by persistent storage. The proxy can acknowledge to clients that writes are finished and stable as soon as the writes are put into the cache. Therefore, it can improves not only read performance but also write performance because overlapping writes can be merged to fewer larger I/Os and then be pushed to the server asynchronously. However, caching's boost to write performance is limited to the period of a file open. Dirty file data in the cache have to be written back to the server when the file is closed. This is required by NFS's close-to-open consistency, which guarantees that when a client opens an NFS file, it can observe the changes made by clients that have closed the file before. Consequently, the write performance would not benefit from the write-back cache when files are closed before any overlapping writes have been merged or any asynchronous write-backs have happened. The overall effect is that writes benefit less from the cache than reads do.



(a) 1M Files

(b) 10M Files

Figure 5.3: Throughput of 16:1 read-write ratio.

The above means that caching boosts performance more for workloads with a higher read-write ratio, which is obvious as shown in Figure 5.3. In fact, we observed a increasing performance speed-up as we gradually increase the read-write ratio from 1 to 2, 4, 8, then 16. Figure 5.4 shows that trend.

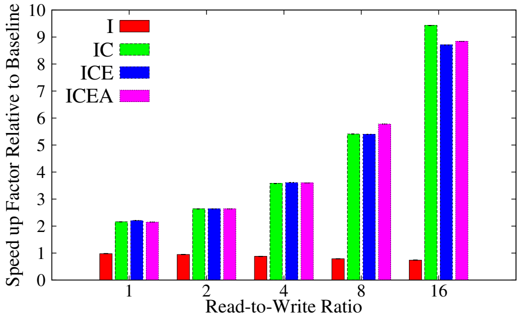
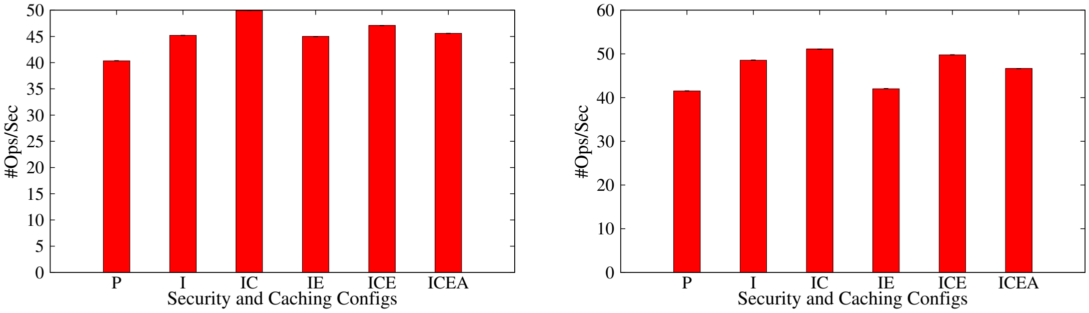


Figure 5.4: Performance speed-up of different read-to-write ratios for 1MB files.

On the other hand, caching does not help the performance as much when the read-write ratio decreases. Specifically, the results of 1:16 read-write ratio is presented in Figure 5.5, where configurations with caching (IC and ICE) outperform their non-caching counterparts (I and IE) by merely 5–18%.



(a) 1M Files

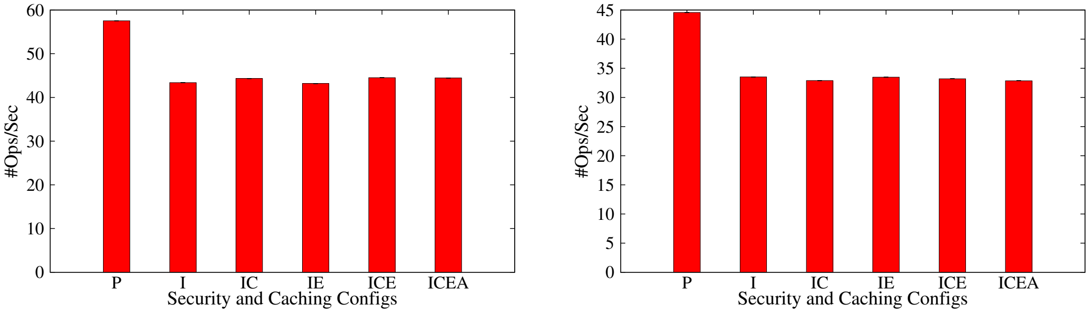
(b) 10M Files

Figure 5.5: Throughput of 1:16 read-write ratio.

Results from the 16:1 (Figure 5.3) and 1:16 (Figure 5.5) read-write ratios reinforce our observations from Figure 5.2 that (1) ICE's performances are only slightly lower than IC's because cached data is cleartext and need no more decryption, and (2) the overhead of antivirus is negligible for small files (1MB), but significant for large ones (10MB).

5.3 Macro-Workloads

This section studies our secure proxy's performance in more complex macro-workloads that more closely match to realistic workloads. We chose the Filebench Mail Server and File Server workloads, which are popular workloads where security features like integrity, encryption, and antivirus are desired.



(a) Mail Server

(b) File Server

Figure 5.6: Performance of Filebench macro-workloads.

The results of Mail Server and File Server (Figure 5.6) are quite similar because these two workloads behave similarly in the secure proxy. First, both workloads contains many small operations (e.g., read, append, and update) wrapped by a pair of file open and close. These open and close operations have performance penalty because many file headers have to be read from the server across the WAN upon open, and all cached dirty data have to be written back immediately upon close. This explains why the integrity configuration (I) performed slower (around 25% for both Mail Server and File Server) than the baseline configuration (P). Second, both workloads have low read-write ratios. This explains why caching does not have a substantial benefit to performance.

We have also benchmarked the Filebench Web Server workload, which has also many small open-read-then-close operations. Compared to the integrity configuration (I), we observe that caching is significantly boosting the performance (by 77%) because of its large read-to-write ratio. However, the overall performance of the secure configurations is still 28% lower compared to the baseline configuration (P). The performance is still lower even when caching are enabled (IC, ICE, and ICEA). This is again because of the frequent open and close operations, which wrap each single small read operation.

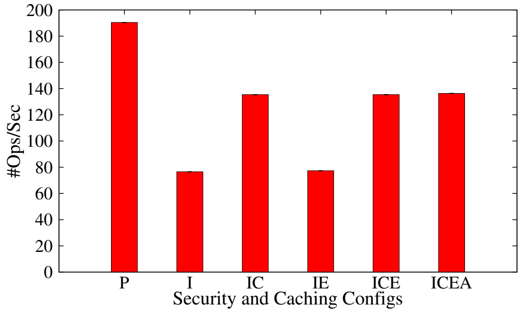


Figure 5.7: Filebench Web Server results

Chapter 6

Conclusions

We presented a study of securing file data stored in cloud. We designed a secure NFS proxy that can provide a rich set of security features to protect clients' data without changing any client or server code. By using stackable file systems [48], our secure proxy has a layered architecture that allows security and performance-enhancing features be added easily and be combined flexibly. We have implemented three security features (i.e., integrity, encryption, and anti-virus) and one performance-enhancing feature (i.e., caching) as stackable file system layers. We have evaluated the performance impact of these features to study the trade-off between security and performance when outsourcing data to cloud storage.

Our secure proxy provides end-to-end integrity, which protects data integrity along the full data path from the trusted proxies to the physical storage device on untrusted servers. We based our design and implementation of the integrity feature on the upcoming technology of Data Integrity eXtension (DIX), which stores a MAC in the out-of-band portion of modern storage devices. The secure proxy protects data confidentiality using efficient authenticated encryption, which encrypts data and generates a MAC at the same time. In addition, the proxy can perform real-time on-access anti-virus scanning, which prevents viruses from spreading across clients. Our secure proxy also provides a persistent caching layer to alleviate the performance overhead of these security features.

We have evaluated our secure NFS proxy with different combinations of features using both micro- and macro-workloads. Our experiments show that our integrity layer can protect data with moderate performance overhead of 3–26%. We observed that adding encryption to integrity introduced only negligible additional performance overhead. On top of integrity and encryption, adding anti-virus cause only negligible additional performance overhead for small files (1MB), but a more significant overhead (up to 23%) for large files (10MB). However, caching is very effective in boosting performance especially for workloads with larger read-write ratios. Because of the caching boost, the overall performance with all four features can be up to 8× of the baseline configuration using a plain proxy.

For all three Filebench macro-workloads (i.e., File Server, Mail Server, and Web Server), we found that integrity incurred significant overhead because these workloads contain a large number of open operations and sent many extra network operations to the remote server. Nevertheless, encryption and anti-virus have only negligible overhead. Also, caching is less effective here because of frequent cache-revalidation and write-back upon file open and close operations. Overall, the secure proxy with all four features performed 23–28% slower than the baseline configuration using plain proxy. This suggests we might need to relax NFS's strong consistency model to further improve performance: for example, allowing a file to be opened without contacting the remote server if its meta-data has been cached recently.

6.1 Limitations and Future Work

The current secure proxy can be improved by adding stronger security protection and by further performance optimization. The current integrity layer is still vulnerable to certain type of attacks. A malicious server still can tamper with a file by substituting its data with data of a different file, or by returning data of a previous version that have already been overwritten. We plan to protect against this swap and replay attacks using a Merkle tree [28] or version numbers in the future. The current encryption layer protects only data but not meta-data. However, meta-data such as file name and directory layout might still leak some important information. We plan to keep file system meta-data also confidential in the future.

Future performance optimizations are also desirable and we are actively working on that. For example, anti-virus can be optimized to scan the file incrementally instead of entirely by breaking common infected files (e.g., PE files) into sections and scanning one section at a time. We also plan to explore the possibility of relaxing NFS's consistency model to reduce the number of round trips to the server. Alternatively, we can try different protocols between the proxy and server such as RESTful protocols.

Bibliography

[1] A. Bessani and R. Mendes and T. Oliveira and N. Neves and M. Correia and M. Pasin and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *USENIX ATC 14*, pages 169–180. USENIX, 2014.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[3] Amazon. *Amazon Simple Storage Service Developer Guide API Version2006-03-01*, 2015. `http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf`.

[4] CBS SF Bay Area. Nude celebrity photos flood 4chan after apple icloud hacked, 2014. `http://goo.gl/p5a49Y`.

[5] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[6] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[7] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, B. Singh, and E. Zadok. Title: Newer is sometimes better: An evaluation of nfsv4.1. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*, Portland, OR, June 2015. ACM. To appear.

[8] Yao Chen and Radu Sion. To cloud or not to cloud?: musings on costs and viability. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 29. ACM, 2011.

[9] Taehwan Choi and Mohamed G Gouda. Httpi: An http with integrity. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[10] Asaf Cidon, Stephen M Rumble, Ryan Stutsman, Sachin Katti, John K Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Usenix Annual Technical Conference*, pages 37–48. Citeseer, 2013.

[11] ClamAV. *Creating signatures for ClamAV*, 2015. `https://github.com/vrtadmin/clamav-devel/blob/master/docs/signatures.pdf`.

[12] P. Deniel. GANESHA, a multi-usage with large cache NFSv4 server. `www.usenix.org/events/fast07/wips/deniel.pdf`, 2007.

[13] Philippe Deniel, Thomas Leibovici, and Jacques-Charles Lafoucrière. GANESHA, a multi-usage with large cache NFSv4 server. In *Linux Symposium*, page 113, 2007.

[14] I/O Controller Data Integrity Extensions. `https://oss.oracle.com/~mkp/docs/dix.pdf`.

[15] Dan Dobre, Paolo Viotti, and Marko Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[16] Yahoo Finance. Cloud computing users are losing data, symantec finds, 2013. `http://goo.gl/x2xnqF`.

[17] John Franks, P Hallam-Baker, J Hostetler, S Lawrence, P Leach, Ari Luotonen, and L Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication. *Internet RFCs*, 1999.

[18] NFS-GANESHA. `http://sourceforge.net/apps/trac/nfs-ganesha/`.

[19] Camille Gaspard, Sharon Goldberg, Wassim Itani, Elisa Bertino, and Cristina Nita-Rotaru. Sine: Cache-friendly integrity for the web. In *Secure Network Protocols, 2009. NPSec 2009. 5th IEEE Workshop on*, pages 7–12. IEEE, 2009.

[20] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145, San Diego, CA, February 2003. Internet Society (ISOC).

[21] iXsystems. FreeNAS. `http://www.freenas.org/`.

[22] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[23] Ari Juels and Alina Oprea. New approaches to security and availability for cloud data. *Communications of the ACM*, 56(2):64–73, 2013.

[24] T. Kojm. ClamAV. `www.clamav.net`, 2004.

[25] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. Safestore: a durable and practical storage system. In *USENIX Annual Technical Conference*, pages 129–142, 2007.

[26] M. Eshel and R. Haskin and D. Hildebrand and M. Naik and F. Schmuck and R. Tewari. Panache: A Parallel File System Cache for Global File Access. In *FAST*, pages 155–168. USENIX, 2010.

[27] Peter Mell and Tim Grance. The NIST definition of cloud computing. Technical report, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2011.

[28] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO'87, pages 369–378, London, UK, 1988. Springer-Verlag.

[29] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004. USENIX Association.

[30] CNN Money. Hospital network hacked, 2014. `http://money.cnn.com/2014/08/18/technology/security/hospital-chs-hack/`.

[31] NetApp. NetApp SteelStore Cloud Integrated Storage Appliance. `http://www.netapp.com/us/products/protection-software/steelstore/`, 2014.

[32] Linux-IO Target, 2015. `https://lwn.net/Articles/592093/`.

[33] End-to-end Data Integrity For NFSv4, 2014.
`http://tools.ietf.org/html/draft-cel-nfsv4-end2end-data-protection-01`.

[34] Arun Olappamanna Vasudevan. Support stacking multiple FSALs, 2014.
`http://sourceforge.net/p/nfs-ganesha/mailman/message/32999686/`.

[35] Data integrity user-space interfaces, 2014. `https://lwn.net/Articles/592093/`.

[36] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[37] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[38] Kapil Singh, H Wang, Alexander Moshchuk, Collin Jackson, and Wenke Lee. Httpi for practical end-to-end web content integrity. In *Microsoft technical report*. Microsoft, 2011.

[39] SoftNAS. SoftNAS Cloud. `https://www.softnas.com/wp/`.

[40] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 229–238. ACM, 2012.

[41] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter USENIX Technical Conference*, pages 191–202, Dallas, TX, Winter 1988.

[42] vinf.net. Silent data corruption in the cloud and building in data integrity, 2011. `http://goo.gl/IbMyu7`.

[43] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, page 19, 2012.

[44] Wikipedia. Sony pictures entertainment hack, 2014. `http://en.wikipedia.org/wiki/Sony_Pictures_Entertainment_hack`.

[45] Network World. Which cloud providers had the best uptime last year?, 2014. `http://goo.gl/SZOKUT`.

[46] SGI XFS. xfstests. `http://xfs.org/index.php/Getting_the_latest_source_code`.

[47] Zadara Storage. Virtual Private Storage Array. `https://www.zadarastorage.com/`.

[48] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[49] E. Zadok, I. Bădulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Monterey, CA, June 1999. USENIX Association.