Security vs Performance: Tradeoffs using a Trust Framework

Aameek Singh[§] Kaladhar Voruganti[†] Sandeep Gopisetty[†] David Pease[†]
Linda Duyanovich[†] Ling Liu[§]

Abstract

We present an architecture of a trust framework that can be utilized to intelligently tradeoff between security and performance in a SAN file system. The primary idea is to differentiate between various clients in the system based on their *trustworthiness* and provide them with different levels of security/performance. Client trustworthiness is evaluated dynamically using a customizable trust model by online monitoring of the client's behavior. We also describe the interface of the trust framework with a block level security solution for an out-of-band virtualization based SAN file system (SAN FS [7]). The proposed framework can also be easily extended to provide differential treatment based on *data* sensitivity, using a configurable parameter of the trust model.

1 Introduction

The designers of security solutions have consistently debated the tradeoffs between levels of security and the resulting performance. Many choose to find a static balance between the two, compromising strictest form of security for better performance. Another practice has been partitioning into groups, with each group having different levels of privileges (security clearance). In such an approach, a group of users (say, within a corporate firewall) might get direct access to data and receive unencrypted transmissions, providing high performance, while another group (outside the firewall) has to authenticate rigorously and can only receive encrypted data, with lower levels of performance. The choice of a solution requires careful planning, analysis of security threats and the sensitivity of stored data.

Consider the case of network attached storage systems, like an out-of-band virtualized SAN file system, IBM SAN FS [7]. In SAN FS, hosts access metadata from dedicated metadata servers (MDS) and access data directly from the storage controllers. For providing block level security in such file systems, capability-based solutions [1, 4, 8, 10] require each access at the storage controller to be validated (encryption/decryption of capability) for correctness. It is conceivable to design solutions where a certain set of clients* are not validated for correctness‡, thus providing them direct access to data for better performance. However, it is essential to design a dynamic framework where clients are provided/revoked this *trusted* access based on their online behavior. For example, a malicious application can attempt exploiting a *trusted* client. A dynamic evaluation system can adjust the trust metric based on the new incorrect behavior and thus dynamically revoke the trusted mode access. Similarly using this mechanism, deploying new clients will not require an assignment to a certain group, rather clients will automatically develop the trustworthiness and thus gain trusted mode access.

In this paper, we present such a trust framework for SAN FS like storage systems. It compromises of (1) a **trust model**, which dictates the metrics used for evaluation of client *trustworthiness* and, (2) **trust distribution** component, which includes the monitoring mechanisms and provides infrastructure required to evaluate the metrics. We also describe the design of a block-level security solution for SAN FS and its interface with the proposed trust framework through a capability-based protocol. Using this combined design, we provide dynamic and configurable trust evaluation, which allows online differential treatment of clients and can be easily extended to configure the model to account for data sensitivity, thus, providing differential treatment based on nature of data being requested.

The rest of the paper is organized as follows. In Section-2, we first briefly explain the design of a capability based block-level security mechanism for SAN FS. Then in Section-3, we describe the proposed trust framework interfaced with the security mechanism. In Section-4 we discuss the related work in security solutions for SANFS-like storage systems and also the use of trust frameworks in other areas like P2P. We finally conclude in Section-5 with a note on future course of work.

2 Block-level Security Design

In this section, we will first briefly describe the design of a capability-based block-level security protocol for SAN FS. This acts as the underlying security solution which will be extended to support the trust framework. Our design is a small variation of various existing protocols [1, 4, 8]. First, we describe the security model assumed for this design.

2.1 Security Model

For our design, we distinguish between network layer security and the application level security. The network

[§]Georgia Tech, {aameek, lingliu}@cc.gatech.edu
[†]IBM Almaden Research, {kaladhar, lduyanov}@us.ibm.com, {sandeep, pease}@almaden.ibm.com
*Ideally, privileges are best granted to applications rather than hosts
‡after network layer security mechanisms for proper identification of such clients

layer security mechanisms prevent against address spoofing, packet sniffing or other network layer attacks like man-in-the-middle. It is a well-researched area and off-the-shelf standards like IPSec are available to ensure network layer security. In addition, we also assume standard hashing techniques like MD5 for ensuring message integrity. In this paper, we only focus on application level security targeted at our application SAN FS. Other than the network layer guarantees, the clients are considered untrusted. The storage controllers and metadata servers are assumed to be trusted and in physically protected environments and share a secret key. We also assume data access rights to be associated with each client application, rather than the client. This prevents buggy (malicious or otherwise) applications from accessing wrong storage. As a result, the authentication/authorization is done at a level of application credential, which can be a secure digital certificate or any other tamper-proof certification. Also, the responsibility of keeping a credential safe is that of the application.

In rest of the discussion, unless specifically mentioned, a "client" refers to an application credential and not a host.

2.2 Secure Protocol

In this section, we give a brief description of our security protocol. We will provide complete implementation details in the extended version of the paper.

Compared to the insecure SAN FS, one of the main modifications in our design is to perform access control checks at the metadata servers (MDS). The MDS should be able to do authentication and access-right checks before giving metadata information to any client. The access can be denied based on both security/access policies and also consistency requirements e.g. another client already holds an exclusive lock on that file. Such authentication and authorization mechanisms can be implemented, based on various known solutions and we call this layer at the MDS - the *authorization server* (AS) layer. Note that it can be implemented outside the scope of the MDS as well, in which case it is contacted before any metadata information is provided to the client. At the storage controller side, we use a security layer called *validator* which is responsible for validating client accesses.

Now, when a client requests metadata from the MDS, the AS checks if access can be granted to the client. If yes, the MDS returns metadata along with a token (capability) of the form:

$$token = K\{ID|EL|AR|TS\}$$

where ID is the unique request ID, EL is the block extent list sent to the client for the request, AR is the access rights for that metadata (read/write), TS is the timestamp at which the token was generated and '|' indicates concatenation. We use $K\{\}$ to indicate encryption using the secret shared between MDS and storage controllers; it can be a symmetric key or an efficient keyed-hash MAC [3].

When the client sends a block request to the storage controller, it includes this token in the request. The security layer at the storage controller decrypts the token and checks if the block requested is included in the blocks contained in the extent list in the token. Also if the client is attempting to write a block, then it has appropriate access rights. If it does, access is granted else denied. This ensures that the storage controller does not provide access to any client that has not been specifically authorized by the MDS.

For every successful request, the storage controller sends back a new token containing an updated timestamp, thus refreshing the token.

$$refreshed\_token = K\{ID|EL|AR|TS_{new}\}$$

where TS_{new} is the timestamp when the request was served by the storage controller.

For revocation of this capability, we use a two-step approach (using ID and TS). We will also analyze revocation mechanisms in prior research [4, 10].

1. Each request gets a unique integer ID from the MDS. Whenever a client gives up a lock, the MDS notifies the storage controller, by sending an explicit revoke message, indicating the ID whose lock has been revoked. Thus, when the client tries to access the storage, the controller first checks if its ID has been revoked by the MDS and deny access if that is the case.

2. To prevent prolonged state maintenance at the storage controller to keep the revocation list, we use a token-expiration mechanism, which automatically expires the token after a certain time (τ). This allows the controller to maintain the revocation list only for τ units of time. It works as follows.

When the client gives up a lock, the MDS notifies the storage controller of the ID whose lock has been revoked. The storage controller keeps the ID in its revocation list for τ units of time. During that duration if the client tries to access the storage controller, the controller rejects the request, based on the ID being revoked. As a result, the client will not be able to refresh its token. After τ units of time, the storage controller dumps the revocation state for the ID and will still deny client request since the TS in its token will be older than τ units and thus, expired.

τ can be set as a system parameter based on workloads - a larger τ requires the storage controller to maintain state for a longer period of time, which is feasible for low load scenarios. The restriction such a mechanism poses is that a client has to generate a request within τ units of time to keep its token refreshed. Notice that pre-fetching of metadata is also handled since the token contains information about all the metadata that a client received from the MDS (and thus has access to) and a successful request for any single block in the metadata will refresh the token for the rest of the metadata as well. In addition, the client can regain access

to the data by getting a new token from the MDS. This can potentially be piggybacked with the data locking and lease renewal mechanisms [7].

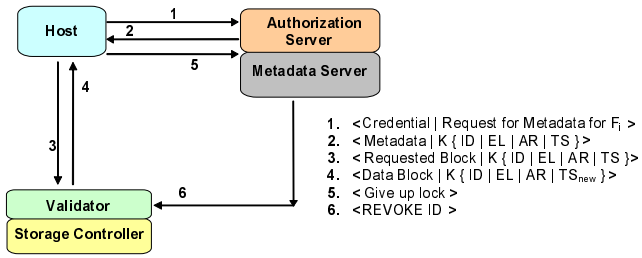

Figure 1: Protocol

Among other research, the revocation has been handled in different ways. [4, 10] use object version numbers, while [1] uses a capability ID like ours and optimizes by grouping capabilities and ability to revoke an entire group. Incorporating grouping in our design, combined with its *self-expiration capabilities* approach, can lead to reductions of revocation traffic and memory utilization at the validator. The complete secure protocol is shown in Figure-1.

3 Trust Framework

In this section, we first discuss our proposed *trusted mode access* mechanism (Section-3.1). Then, we discuss the kind of trust ratings/models that we use. We will also describe variations to our proposed infrastructure and tradeoffs involved between them.

3.1 Trusted Mode Access

In our trust infrastructure, we associate a trust rating with each application (client credential) and store that information at the MDS. Now when a client requests metadata from the MDS, the MDS checks the trust rating based on its credential and then according to a policy (as described in Section-3.2) can decide to trust the client. The objective now is to prevent the encryption and decryption costs at the storage controller for each data block request, as in the original design, and let the storage controller service all requests for the client without checking the validity.

This is accomplished using another new message between the MDS and the storage controller. Specifically, the MDS will send a

$$< TRUST\ Credential >$$

message to the storage controller, which indicates to the controller that the client is fully trusted to access the correct storage and thus whenever the client requests certain blocks, the controller should just fulfill the request, without decrypting the token. It also does not need to refresh the token (preventing encryption). Also, the credential is assumed to be trusted until the MDS sends an explicit

$$< REVOKE\ Credential >$$

message to the controller.

To prevent clients with good trust ratings from exploiting this mode of access, we use a strict trust model and an auditing mechanism described in Section-3.2.

We believe that this trusted access mechanism can have a significant performance impact, since in any enterprise setting, there will potentially be a number of applications that are completely trusted or have access to all storage (monitoring applications, compliance applications). Those applications can always operate in such a trusted mode, thus eliminating the need of validating tokens, which requires expensive cryptographic operations. From the storage controller point of view, it only has to keep an additional state of all trusted credentials, which is not a prohibitive overhead.

Given the general description of how we enable trusted mode access, we discuss the details of our trust infrastructure next.

3.2 Trust Infrastructure

The use of trust ratings has been extensively researched in reputation based IR systems, recommendation systems, P2P systems and other ecommerce settings [6, 9, 11]. There are two key components of any trust-based infrastructure:

- **Trust Model**: This determines the model of trust associated with each client, for example, whether any client has only a binary trust rating - 1 indicating trusted and 0 indicating not trusted or a continuous rating in $[0, 1]$, 1 being most trusted and 0 being least trusted. It also determines how a measurable metric is mapped to a trust rating, for example, if client accesses right storage more than 80% of time, it is trusted (trust rating 1 for the former model), or trust rating is equal to the percentage of correct access (trust rating 0.8 for the latter model).

- **Trust Distribution**: This component is responsible for providing the infrastructure that is required to evaluate the metrics, used by the trust model to compute the rating. For example, how to calculate number of successful transactions (for the example models above) and how to disseminate this information to appropriate agents, which act on these ratings.

First we discuss the trust model component of our design.

3.2.1 Trust Model

For the purpose of this discussion, assume that the trust ratings are somehow available to the MDS. We explain in the trust distribution component of how that is achieved. Also, for the rest of the discussion, a transaction refers to an access of storage at the storage controller.

In our design, we use a [0,1] trust model with each client having a trust rating from 0 to 1. The trust rating is dynamic and changes with the behavior of the client. We also set the probability of a client getting a trusted mode access equal to

its trust rating, e.g. a client with trust rating of 0.6 has a 60% chance of getting a trusted mode access and it retains the access until its credential is revoked. The revocation can occur when it no longer has access to the metadata it requested (e.g. if it gave up the lock) or its trust rating drops (the exact policy in this case is described below).

In addition, the client applications have different trust values for each storage controller. This is to prevent a buggy, but not malicious, application that accesses its storage controller in a correct manner but incorrectly accesses another storage controller, from being penalized on its correct accesses. Also this potentially helps us in doing application error detection. However, this approach may increase the trust ratings store size and may make it prohibitive when there are a large number of client applications. In those scenarios, we recommend a single rating across all controllers.

As discussed earlier, we want to have a strict trust model, so that clients are strongly discouraged from accessing wrong storage. Also, it must be relatively tough to build a good trust rating when starting from scratch. This is required to prevent a malicious application to start afresh and gain a good trust rating easily and then exploiting it. We achieve both of these requirements in the following manner. First, we set a threshold ψ, on the total number of transactions done before a client can ever be allowed to operate in the trusted mode. After that threshold is achieved, the clients probability of getting a trusted mode access is proportional to the ratio of the correct transactions to the total number of transactions. Specifically,

$$\text{Pr(Trusted Access)} = \text{Trust Rating} = \begin{cases} 0, & \#tr < \psi \\ (\frac{\#ctr}{\#tr})^{\alpha} & \#tr \geq \psi \end{cases}$$

where #tr is the total number of transactions and #ctr is the number of correct transactions and $0 \leq \alpha \leq 1$ is a configurable parameter determining the strictness of the model desired.

The threshold ψ prevents applications from gaining a good trust rating immediately. After that, the probability of getting a trusted mode access is determined by the probability of the access being correct (equal to the ratio of correct transactions to the total number of transactions). It can be argued that the threshold is achieved by just doing a large number of transactions, irrespective of the correctness. However, it is highly likely that a malicious/buggy application can be detected before the threshold is achieved, and in addition, if those transactions were largely incorrect, the probability of getting a trusted mode access will be very low (which can be further penalized by setting low values of α).

As discussed above, once a client gets trusted mode access, it retains access until it is specifically revoked by the MDS. We set this revocation policy as follows. Whenever a client accesses the wrong storage, its trust value drops (because the ratio drops) and that indicates the MDS to revoke its trust credential (if it is in trusted access mode). The mechanism of identifying a wrong access is detailed in the trust distribution component of the architecture.

Extensions

Note that till now the trust rating is only a function of client behavior. However, in our trust model, it is easy to adjust the client trust rating for a storage controller based on the kind of data stored in that controller. For example, if an organization stores extremely critical data at a particular storage, it can ensure that the trusted mode access is not allowed or extremely difficult to get for that storage controller. This can be achieved by simply setting a very small value of α (α=0 means no trusted mode access except for absolutely perfect, #ctr = #tr, applications). This provides an easy extension by incorporating differential treatment based on data sensitivity, in our design.

Another possible extension is to provide different levels of trusted mode access. For example, for a moderately trusted application, we can use smaller keys in encryption of the capability to provide a better level of performance, but still with more security than a complete trusted access.

3.2.2 Trust Distribution

Now, we discuss the trust distribution component of our infrastructure. This details the mechanisms required to obtain the information necessary to compute trust ratings.

Given the above trust model, MDS requires statistics about the number of transactions and the number of correct transactions for each client at every storage controller. During a non-trusted mode access, the security layer at the storage controller can easily compute these numbers. Both #tr and #ctr are maintained as counters, with #tr incremented for every access and #ctr incremented if the access was granted after validating the token.

On the other hand, when a client accesses storage in the trusted mode, the token is not decrypted and thus it can not be immediately ascertained that the access was correct or not. In this case, we use an auditing mechanism. Note that even in trusted mode access, the MDS *does* give a valid token for the first time when the client requests the metadata. In order to catch any violations during the trusted mode access, the security layer logs the encrypted token along with the requested blocks information. An auditing process will decrypt the token at a later time and deduce whether the client accessed the right storage. Thus, while in non-trusted mode the overheads are due to a decryption and an encryption, the trusted-mode access has the overhead of logging (extra writes for logs) and auditing which are amortized by the total number of requests. We believe that this mechanism will still reduce individual response times.

In addition, we can extend this auditing mechanism to be a probabilistic mechanism as well. For example, during a trusted mode access, a sample of all accesses is actually logged, thus reducing the overheads. The size of the sample can be further determined based on client behavior, for example, the number of times the client accessed wrong storage during a trusted mode access (indicating malicious behavior).

We continue to investigate this and will present an efficient mechanism in the detailed version of the paper.

The trusted mode access logs have the following structure:

$$< Credential \mid Token \mid TS_1 \mid Block\text{-}1, Block\text{-}2, \ldots >$$

where the credential is the application credential in the trusted-mode access. The token is the first token, the storage controller received from the client when the trusted mode access was initiated, TS_1 is the timestamp of a first data request using this token and the blocks are the addresses of data blocks accessed by the client during the trusted-mode access. The TS_1 entry of the log is to prevent the scenario where a malicious client sends an expired token and tries to access the blocks allowed under that token, even after the access rights were revoked*. The auditing process later decrypts the token and verifies that the blocks accessed were allowed under the access rights of that token. Note that it is possible for a client to use multiple tokens within a single trusted-mode session. In that case all those tokens are logged.

The auditing process can either be located at the storage controller using its free CPU cycles or on a different server (possibly at one of the MDS) which can access the trusted-mode access logs to update the counts. The metadata servers then update the trust ratings in batches periodically. Thus, the trust rating is not modified after every client transaction, rather modified in batches. This is done to prevent excessive communications between the controller and the MDS. If the auditing process is located at the MDS, the total overheads are reduced, since the trust ratings can be evaluated during the auditing process itself.

It is possible for the storage controller to store the trust ratings at the controllers itself and probabilistically decide whether to verify the token or not. The reason we chose the MDS to store trust ratings was to have flexibility in controlling the trust process. For example, it would be easy to update the trust model, change the values of ψ and α parameters or change the policy for granting trusted-mode access. Also it is easier to perform static trust settings, for example an administrator can explicitly specify an application to be trusted (say, a monitoring application which has access to all storage) without waiting for it to gather a good trust rating. Using MDS hosted trust value, we have eliminated the need of a controller to have any knowledge of trust ratings.

4 Related Work

There has been significant amount of work in security of out-of-band virtualized SAN file systems [1, 4, 8, 10]. These solutions are broadly all capability-based solutions and differ in details of capability design and revocation mechanisms. Another approach towards security has been through the Object Store initiative [5, 2], in which data is assumed to be accessed in the form of objects with each object having associated access rights with it. All of these solutions attempt to provide secure protocols for complete security under their assumed models. Our work is distinguished from the prior research due to the tradeoff mechanisms in which we choose to differentiate between clients and the data being requested, and give different levels of performance. The tradeoff mechanism is designed in the form of a trust framework, utilizing the trust related work in P2P, and ecommerce [6, 9, 11].

*If the malicious application accesses blocks outside the extent list in the token, it will be captured by the auditing process

5 Conclusions and Future Work

In this paper, we presented a trust framework which is used to provide different levels of security and performance to distinct clients based on their system behavior. The trust model is customizable and dynamic to automatically promote and revoke the levels of access to the clients. In addition, the model can be easily extended to provide different levels of security based on the nature of data being requested. We also presented the interface of the trust framework with a block-level security solution for a SAN file system. In the future, we plan to empirically evaluate the framework design for various benchmarks and also increase the efficiency of the trust distribution component.

References

[1] M. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. Thekkath. Block-level security for network-attached disks. In *Proc. FAST*, 2003.

[2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *Proc. MSST*, 2003.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Lecture Notes in CS*, 1109:1–15, 1996.

[4] H. Gobioff, G. Gibson, and D. Tygar. Security for network attached storage devices. *Technical Report CMU-CS-97-185*, 1997.

[5] T10 http://www.t10.org. InterNational Committee on Information Technology Standards (INCITS), 2004.

[6] Z. Despotovic K. Aberer. Managing trust in a p2p information system. In *Proc. CIKM*, 2001.

[7] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank - A Heterogeneous Scalable SAN file system. *IBM Systems Journal*, 42(2), 2003.

[8] E. Miller, W. Freeman, D. Long, and B. Reed. Strong security for network-attached storage. In *FAST*, 2002.

[9] A. Rahman and S. Hailes. Supporting trust in virtual communities. In *Proc. HICCS*, 2000.

[10] B. Reed, E. Chron, D. Long, and R. Burns. Authenticating network attached storage. *IEEE Micro*, 20(1), 2000.

[11] L. Xiong and L. Liu. A reputation based trust model for peer-to-peer ecommerce communities. In *Proc. Electronic Commerce*, 2003.