A FAULT-TOLERANT KEY STORAGE SERVICE WITH PROACTIVE RECOVERY

By

PUNIT AGRAWAL

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and
Computer Science
August 2012

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation/thesis of PUNIT AGRAWAL find it satisfactory and recommend that it be accepted.

_____
Dr. Carl H. Hauser, Chair

_____
Dr. David E. Bakken

_____
Dr. Min Sik Kim

ACKNOWLEDGEMENT

This research project would not have been possible without the support of many people. I wish to express my gratitude to my supervisor, Dr. Carl Hauser who was abundantly helpful and offered invaluable assistance, support and guidance. I would also like to thank Dr. David E. Bakken and Dr. Min Sik Kim for serving in my committee and reviewing my thesis.

I also take this opportunity to acknowledge my parents Mr Dinesh Agrawal and Mrs Veena Agrawal, my sister Leena Bhai, my brother-in-law Siddharth Bhai and my friends for being a great support and making this happen. I am also very thankful to my lab mates Yujue Wang and Kelsey Cairns for reviewing my thesis and to Dave Anderson for his help with GridStat. Lastly, I am thankful to National Science Foundation (NSF) for their continued funding.

A FAULT-TOLERANT KEY STORAGE SERVICE WITH PROACTIVE RECOVERY

Abstract

By Punit Agrawal, M.S.
Washington State University
August 2012

Chair: Dr. Carl Hauser

A lot of security architectures for digital systems have been developed that rely on the use of cryptographic keys, to provide their functionality. The distribution of symmetric keys between the encryption and decryption entities has always been a matter of concern, and there is no easy way to do this while maintaining the confidentiality and integrity of the keys. This thesis provides a service which while ensuring the confidential storage of cryptographic keys, also ensures that the system is tolerant to an unlimited number of failures, given that only a threshold number of failures occur during a defined window of system execution. The service also attempts to improve performance and availability by modeling a Byzantine Quorum system.

A potential application of the system is also shown by integrating the service with GridStat. GridStat is a middleware framework based on the publish-subscribe paradigm that provides flexible, robust, timely, and secure delivery of operational status information for the electric power grid. The system has been evaluated mostly with performance in mind and results show promising latencies in the operations provided by the service.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE - INTRODUCTION

1.1 Motivation

The storage of confidential information has been researched extensively especially in the last decade as a result of increased reliance on digitally stored information. Accordingly, the availability, integrity and confidentiality of this information need to be provided. The computing resources of the digital devices and their storage capacities are also constantly being upgraded, giving the researchers more options to develop storage systems in which users can store private information with the guarantee that it is kept confidential, is continuously available and cannot be destroyed. However, this trend has also given more power to the attackers to compromise the same systems.

Wylie et. al. [27] introduced a similar concept of survivable systems in their PASIS architecture. The underlying principle that they followed was that no individual service, node or user can be trusted and survivable systems must replicate and distribute data across multiple nodes, with no single node capable of leaking the confidential information. Also, the system must continue to operate correctly despite the occurrence of accidental and malicious faults (including security attacks and intrusions). These requirements are especially relevant in the case of systems used to manage critical infrastructures like the Power Grid. The Power Grid itself has been designed with the $N-1$ fault-tolerance principle in mind, meaning that it is built to survive at least one failure. This redundant design is such that if a power asset such as a transmission line or

generator were to fail the power would be routed from elsewhere without causing major blackouts.

There has been an increasing use of electronic controls and automation technologies in the U.S. power grid during recent decades. This implementation, termed as Smart Grid [31] has a big effect on the grid, changing the way it operates, communicates, and ultimately delivers power. The Smart Grid gives the operators more visibility into the real-time behavior of the grid. However it also increases the importance of protecting the availability, integrity and confidentiality of system data, since access to this detailed operational data can be valuable to adversaries that intend to make the grid unstable. With a smart grid, customers also have an energy services interface (ESI) that communicates upstream to the utility through the advanced metering infrastructure (AMI) [31], which empowers them to better manage their electricity use. Again, this two-way information exchange presents new cyber security challenges to protect data security and customer privacy. The key-storage service discussed in this thesis complements the existing solutions that have been proposed to address the cyber-security challenges in the power grid.

1.2 Scope of thesis

This paper discusses the design of a fault-tolerant distributed service for storage and dissemination of confidential information **(*ProFokus*).** Our work derives a lot from the CODEX system [15], and the COCA framework [28], that offer similar functionalities. This system, like CODEX is designed to be used in a publish-subscribe middleware infrastructure like GridStat [22] , providing support for the storage and dissemination of publication keys, used to secure the published variables. However, our protocols unlike

CODEX, offer the mechanism to update the secret information associated with a particular name and to store keys of any length.

The $ProFokus$ service is coupled with Proactive Recovery to make it more resilient. The aim of the Proactive Recovery mechanism is conceptually simple – components are periodically rejuvenated to remove the effects of malicious attacks/failures. If the rejuvenation if performed frequently often, then an adversary is unable to corrupt enough resources to break the system. Proactive recovery has been suggested in several contexts. For instance, in COCA, it has been used to refresh cryptographic keys in order to prevent the disclosure of too many secrets [29]. It has also been utilized to restore the system code from a secure source to eliminate potential transformation carried out by the adversary. It may include substituting the programs to remove vulnerabilities existent in previous versions (e.g., software bugs that could crash the system or software errors exploitable by outside attackers).

The effectiveness of proactive recovery protocols implemented in the aforementioned CODEX system depends on two primary assumptions: periodic and timely execution. They assume that the proactive subsystem is regularly executed, and that the rejuvenation operation does not take a very long period to complete. In other words, these proactive recovery protocols make timing assumptions about the environment, which by definition can be violated in an asynchronous setting.

It has been proved in literature that it is theoretically impossible to have permanently correct f fault-tolerant asynchronous systems, i.e., asynchronous systems which can only tolerate a bounded number of faults [30]. These include systems that use asynchronous proactive recovery. The description of a possible attack to CODEX and

COCA that is based on a time-related vulnerability of the proactive recovery protocol it uses [23]. COCA embraces the asynchronous communication model, however also implicitly assumes that it has access to a clock with a bounded drift rate. This requirement is necessary to allow the periodic and timely triggering of its asynchronous proactive secret sharing protocol (APSS) [29]. But, by definition, in an asynchronous system no such bounds exist [9]. Moreover, if an attacker gains access to the clock, she or he can arbitrarily change its progress in relation to real time. To counter these problems, a different proactive recovery approach using a more synchronous system has been proposed in this thesis.

1.3 Research Contributions

The research contributions of this thesis are:

- Design of a fault-tolerant key-storage service with proactive recovery, that ensures the confidentiality, integrity and availability of the stored keys
- Implementation of the key-storage service in JAVA
- Integration of the key-storage service with a publish-subscribe middleware called GridStat
- Experimental performance evaluation of the service's protocols.

1.4 Organization of Thesis

This thesis is divided into six chapters. Chapter 1 introduces the problem space and the scope of the thesis. Chapter 2 delves into more background details regarding the concepts used to implement the service. Chapter 3 presents the design of the service and its protocols in full detail, while Chapter 4 gives details about the proactive recovery

mechanism implemented in the service. Chapter 5 gives an example of an application of the service, by integrating the service with a middleware framework called GridStat. Chapter 5 also details the changes to be made to GridStat to support $ProFokus$. The implementation of the key service and the performance of its protocols are discussed in Chapter 6. Finally, Chapter 7 offers concluding remarks and potential areas of future research.

CHAPTER TWO – BACKGROUND

This chapter gives an overview of the various concepts that will be used later to describe the system.

2.1 Fault-tolerance

A service is said to be fault-tolerant if it continues to operation according to its specifications, despite the failure of some components. The design of a fault-tolerant system depends on the type of failures that can occur and their number. Some common failure types for components include:

- Crash failure: When a component experiences a crash failure, it halts prematurely. The component behaves correctly, until it halts.
- Omission failure: When a component experiences an omission failure, it fails to send or receive any messages. Crash failure is a special kind of omission failure.
- Byzantine failure: When a component experiences Byzantine failure, it may deviate arbitrarily from its specification [13].
- Disclosure failure: When a component experiences this failure, any information stored at the component is leaked to an adversary.

Byzantine failures are the most difficult to tolerate and so is the design of a Byzantine fault-tolerant service. However, such a service is desirable since it does not make any assumptions regarding the behavior of faulty components. Malkhi and Reiter [14] proposed the use of quorum systems for tolerating Byzantine failures and this technique is discussed next.

2.1.1 Byzantine Quorum System

Byzantine Quorum Systems [14], hereafter called simply as quorum systems, have been used to implement replicated data storage systems, ensuring consistency and availability of the stored data even in the presence of Byzantine faults in some servers. Quorum systems algorithms are recognized for their good performance and scalability, as each request is handled and processed by only a subset of the servers instead of all servers. This subset of the servers is called as a quorum. Any two quorums have a nonempty intersection that contains a sufficient number of correct servers (ensuring consistency). Also, there is at least one quorum in the system that is formed only by correct servers (ensuring availability).

The size of the quorum and the total number of servers required are dependent on the type of quorum system used which itself is dependent on whether the data is *self-verifiable* or not. *Self-verifiable* data is a type of data that faulty servers can fail to redistribute but cannot undetectable alter. If the data is *self-verifiable*, then a *dissemination* quorum system can be used, otherwise an *opaque masking* quorum system needs to be used. The quorum sizes and the number of servers required by these types of quorum systems to handle t Byzantine failures are shown in Table 1.

Type of Quorum system	Number of servers n	Quorum Size $\lvert Q \rvert$
Opaque Masking	$n \geq 5t$	$\left\lceil \frac{2n + 2t}{3} \right\rceil$
Dissemination	$n \geq 3t + 1$	$\left\lceil \frac{n + t + 1}{2} \right\rceil$

Table 1 – Types of Byzantine Quorum systems

In case of *dissemination* quorum systems, the number of servers, $n \geq 3t + 1$ since the intersection between any two quorums would then be at least $t + 1$ and thus will

contain at least one correct server and hence the stored data can be obtained from this server, since the data is *self-verifiable*. In case of *opaque masking* quorums, the intersection between any two quorums must be increased and a single correct server in the intersection will no longer suffice since the data is not *self-verifiable*. So the intersection needs to be more number of correct servers than the number of faulty servers that can reply so that voting can be employed to get the correct response. Moreover, since these faulty servers can "team up" with the out-of-date but correct servers, the number of correct servers that the intersection must contain, must be no less than the number of faulty or out-of-date servers that can reply [14].

The $ProFokus$ system presented in this thesis is used to store the cryptographic keys and the keys are not self-verifiable and cannot easily be made so. This is because the values are just the public encryption of the confidential information and hence everyone who has access to the $ProFokus$ public key (every process in the system) can create the values. Hence, $opaque\ masking\ quorum\ systems$ [14] are employed by $ProFokus$ and consequently, the minimum number of servers required in $ProFokus$ to tolerate t faults is $5t$, with the quorum size being $\left\lceil \frac{2n+2t}{3} \right\rceil$.

2.1.2 Proactive Recovery

The use of the Quorum System described in the previous section enables the $ProFokus$ system to tolerate up to t failures during the execution of the system. This requires the system administrator to predict a priori the maximum number of failures that can occur during system execution. As we will learn later in Section 3.1, the $ProFokus$ system embraces the asynchronous system model, where there are no bounds on server

execution speeds and message delay times. Consequently, the system execution time becomes unbounded. Hence, it becomes mathematically infeasible to predict the maximum number of failures that can occur during an unbounded system execution time. This problem has led to research related to proactive recovery and has made it possible to develop asynchronous systems [15][28] that can tolerate any number of failures over the lifetime of the system, provided fewer than a predefined number of failures occur during a small window of vulnerability. This is accomplished through the use of proactive recovery protocols [24,25] that periodically rejuvenate the system. The details of the proactive recovery protocols employed by the $ProFokus$ system will be described in the next chapters.

2.2 Key Confidentiality and Integrity

The fault-tolerant approaches described above help to achieve the required level of availability for the service; however, the service, while being highly available needs to protect itself from malicious adversaries. There are majorly two models of adversaries:

Passive Adversary: A *passive adversary* only learns the information stored at a server, but is unable to alter the behavior of the server

Active Adversary: An *active adversary* learns the information stored at a server and also has the ability to alter the behavior of the server. Such an adversary is also called a *Byzantine adversary*.

We assume the active adversary model, analogous with the Byzantine fault tolerance of the $ProFokus$ systems. The adversary is able to attack any of the components in the system, which include:

- the servers that are responsible for handling requests, sending out responses and storing the client key-values, and
- the communication channels between the servers and between the servers and the clients.

In case of the servers, the adversary can steal the information like the key-values stored at the servers, and can alter the server behavior, which may cause the server to respond maliciously to requests. In case of the communication channels, the adversary can read the messages sent on the channels and the key-values contained in it, and also insert, remove or modify messages on the communication channels. The correct functioning of the *ProFokus* system depends on ensuring that the following properties hold:

- **Confidentiality**, which ensures that the key-values are only disclosed to authorized entities
- **Integrity**, which ensures that key-values are not modified while in transit
- **Authenticity**, which allows the receiver of a message to verify the origin of the message.

The methods used to ensure these properties are discussed below.

2.2.1 Encryption

The primary way to achieve confidentiality is through Encryption, which is a cryptographic process to transform plaintext into ciphertext, based on an encryption algorithm and a key. The plaintext can be deduced using a decryption algorithm and the same key. Two classes of cryptographic algorithms [26] are present:

- **Symmetric key cryptography:** In this, the encryption and decryption algorithms operate on the same key
- **Asymmetric key cryptography:** In this, different keys are used for the encryption and decryption algorithms. The key used for encryption is termed as the public key, while the key used for decryption is termed as the private key. The private key is kept confidential and is never revealed. The public key on the other hand can be published to anyone who needs to do the encryption.

Asymmetric key cryptography offers functionality beyond the capabilities of symmetric key cryptography. For instance, asymmetric key cryptography can be used to implement digital signatures [26] which ensure the authenticity and integrity of the data being signed. An entity E_1 can generate a digital signature for a message by invoking a *digital signature algorithm* using its private key K_{S1}. Another entity E_2 with knowledge of the public key K_{S1} can verify the integrity of the message by applying a *signature verification* algorithm to the signature. Because private key K_{S1} that is used to generate the signature is known only to entity E_1, no one except E_1 can produce such a signature. Any entity who knows K_{S1} can verify the signature and be convinced that the message was signed by E_1 and has not been subsequently altered by others.

Given the benefits, the $ProFokus$ system makes use of asymmetric key cryptography to ensure the confidentiality and integrity of key-values stored at the servers and to ensure the authenticity of request and response messages. The system maintains a service public-private key pair. The private key is kept confidential, while the public key is published to all the clients. More details will be discussed when we describe the complete protocols in Section 3.2

2.2.2 Threshold Cryptography

Asymmetric key cryptography helps to achieve confidentiality but the confidentiality of the service private key still needs to be handled. Since each server in the system is prone to Byzantine failures, no server can be trusted with the key. So the $ProFokus$ system employs $(n, t+1)$ threshold cryptography [26] to distribute the private key to all the servers in such a way that:

- The key shares reveal no information about the private key
- At least $t+1$ servers are required to sign a message or to decrypt a value encrypted with the corresponding public key.

The main goal of threshold cryptography is to perform the procedures of digital signature or decryption without ever reconstructing the private key. The steps involved in a $(t+1, n)$ threshold signature scheme are as follows:

- In the initial stage, a trusted dealer initially generates n secret key shares $(s_i, i \in \{1, .. n\})$, n verification keys $(v_i, i \in \{1, .. n\})$, the group verification key v and the public key PK used to validate signatures.
- Next, the dealer sends these keys to n different servers, alternatively called as *shareholders*. Thus, each *shareholder* i receives its partial key s_i and its verification key v_i. The public key and the n verification keys are available to every server in the system. After this initial setup, the system is able to generate signatures.
- To obtain a signature A to some data d, each *shareholder* i generates its partial signature a_i (also called signature share) of d. Later, a combiner obtains at least $t+1$ valid partial signatures $(a_1, \ldots a_k)$ and builds the signature A through the

combination of these $t + 1$ valid partial signatures. An important feature of this scheme is the impossibility of generating valid signatures with $t\ or\ fewer$ valid partial signatures.

The above scheme is based on the following primitives:

- $ThreshSign(s_i, v_i, v, x)$: Module used by the shareholder i to generate its partial signature a_i for data x along with a "proof of correctness" ρ_i
- $ThreshVerify(x, \rho_i, PK, v, v_i)$: Module used to verify if the partial signature a_i, obtained from the shareholder i, is valid for data x
- $ThreshCombine(a_i, .., a_k, PK)$: Module used by combine signature A from k valid partial signatures
- $verify\ (A, x, PK\ )$: Module used to verify if A is a valid signature of data x.

A $(n, t + 1)$ threshold decryption scheme works in a manner analogous to the threshold signature scheme, except that instead of generating a signature, it results in the decrypted value of the ciphertext. The use of threshold cryptography ensures that the private key is stored securely and an adversary must compromise more than t servers to learn the private key. This $t - fault - tolerance$ is closely tied with the fault tolerance threshold, presented in Section 2.1.

2.2.3 Blinding

Blinding [26] is a concept in cryptography that allows a client to have a server compute a mathematical function $y = f(x)$, where the client provides an input x and retrieves the corresponding output y, but the server would learn nothing about neither x

nor y. This concept is useful if the client cannot compute the mathematical function f all by itself.

For instance, a server can offer to decrypt a ciphertext x using a decryption function D and some private key K_p which is not known to the client. The blinding process then works as follows:

1) The client generates a blinding factor b_c and calculates blinded ciphertext $m' = m * b_c$
2) The client sends m' to the server; the server applies decryption function $D(m', K_p)$ to obtain the plaintext x' and sends it to the client.
3) The client divides the obtained plaintext x' by the blinding factor b_c and the result is the required plaintext

For the above blinding process to work, it is required that the cryptographic scheme used be $blindable$. A cryptographic scheme is said to be $blindable$ if its encryption function E and decryption function D are both homomorphic, i.e.

$$E(a * b) = E(a) * E(b) \; and$$

$$D(a * b) = D(a) * D(b)$$

RSA [17] is an example of such a blindable cryptographic scheme.

CHAPTER THREE - PROACTIVE FAULT-TOLERANT KEY STORAGE

This chapter discusses the design of the Proactive Fault Tolerant Key Storage System ($ProFokus$) in terms of the assumptions made about the environment in which the system operates, the services that are being offered and the Protocols that are used to implement the services.

3.1 System Model

The $ProFokus$ system consists of a set $S = \{S_1,\ S_2,..S_n\}$ of n servers, each running on a separate processor in a network and having unique identifiers. The system is built to handle failures subject to the following assumptions:

Compromised Servers: The servers are executing either in a correct state or a compromised state, where a compromised state denotes a state in which the server might have experienced a Byzantine failure and/or might have disclosed information locally stored in it. A process that exhibits Byzantine failure may show any behavior: it may crash, remain silent and ignore requests and/or send arbitrary responses. The system execution is divided into time periods called windows of vulnerability. We will discuss these time periods in detail when we talk about proactive security later in this section. In each such window, a server is deemed correct" or "compromised". A server is deemed correct if and only if that server has not been compromised throughout the entire period, otherwise it is deemed as compromised. We assume that at most $\boldsymbol{t}$ of the $\boldsymbol{n}$ $ProFokus$ servers are compromised during each window of vulnerability, where $\boldsymbol{n \geq 5t}$ holds.

Furthermore, we assume that the servers are *fault-independent*, i.e. the failure of one server process does not affect any other server process in the system, nor does it reveal any vulnerability about any other server process. One way to achieve *fault-independence* is to employ diversity [5] in the software that the servers run, so that the $ProFokus$ servers are not identical in their implementations and have no common vulnerabilities.

Insecure Links Assumption: The $ProFokus$ servers use an unreliable and insecure channel to communicate with each other. Messages in transit may be disclosed to, deleted or modified by an adversary. Also, new messages may be inserted by an adversary. However, if a server sends infinite times a message to another correct server, then this message will eventually be delivered.

Asynchrony Assumption: The message delivery delays between the servers and between the clients and the servers are potentially unbounded and there is no bound on the server execution speed as well. This means that an adversary can employ a denial-of-service attack that may delay messages in transit or slow the processing at the servers by arbitrary finite amounts.

3.2 Services Provided by $ProFokus$

The $ProFokus$ system provides functionality to store the name to key bindings. Only one key is bound to a name at a point in time; however the bindings can be updated to specify new bindings. The various operations that $ProFokus$ implements to support this functionality are:

- ***Update :*** to create or update a new name-key binding
- ***Read_Key :*** to read the key bound to a particular name
- ***Read_Timestamp :*** to read the timestamp bound to a particular name

The three operations can be invoked by different clients. For example, a client can invoke an *update* operation to create a name-key binding and some other client can execute a $read\_key$ operation to retrieve that binding. The $ProFokus$ system provides the following guarantees to its clients:

- **Availability:** When authorized clients invoke any of the above operations for a particular name and such invocation is not concurrent with any other invocation for the same name, then the requested operation will be performed if the invocation request is sent sufficiently often.
- **Confidentiality:** The only way to learn the key associated with a particular operation is to execute the $read\_key$ operation. Only authorized clients that have successfully executed the $read\_key$ operation for a particular name have knowledge of the key associated with that name, given that the Compromised Servers assumption stated earlier is not violated.
- **Integrity:** The only way to create/update a name-key binding is to execute an update operation for that particular name.

To invoke any of the above operations, a client c sends a request to the $ProFokus$ system and awaits a response. The $ProFokus$ servers expect each client request to contain a nonce and to be digitally signed by the client. Ill-formed invocation messages are ignored by $ProFokus$. The response received from the $ProFokus$ system is digitally

signed using a service key and contains the original request sent by the client. This enables the client to check the response and ascertain that the response is not a replay.

Each binding β stored at the $ProFokus$ servers is a tuple $< \tau, N, k >$ where τ is the timestamp associated with the binding, N is the name and k is the stored secret key. The timestamp τ is assigned by the $ProFokus$ servers in increasing order, depending upon the update request sent by the client. Each $ProFokus$ server assigns a default timestamp of zero to any binding that does not yet exist in its database. Given this nomenclature, the semantics for the $ProFokus$ operations can be given as follows:

i. $Read\_key$: Given a name N, a $read\_key$ request returns a binding β such that the binding β was created by the most recent successful update request.

ii. $Read\_timestamp$**:** Given a name N, a $read\_timestamp$ request returns the timestamp τ such that the timestamp τ was sent in the most recent successful update request

iii. $Update$**:** Given a timestamp τ and a new value k for name N, an $update$ request returns an acknowledgement after the $ProFokus$ system has successfully created a new binding $\beta' =< \tau + 1, N, k >$.

3.2.1 Server Key Pair

In the $ProFokus$ system, each server maintains a public/private key pair, with the public key known to all other servers. These public keys allow the servers to authenticate the sender as well as check the integrity of the messages that they exchange with other servers. The clients do not need to have any knowledge of the public keys of the $ProFokus$ servers and this feature is very useful in a system with a multitude of clients

and this allows server keys to be changed during proactive recovery without having to notify the clients. Currently, however, we assume the presence of a tamper-proof secure crypto-processor which stores the private key of each server. This crypto-processor also functions as a black-box to perform digital signage of the messages that it receives and hence, the private key is not revealed in this process. Because of this assumption, we do not refresh the server keys as part of proactive recovery (discussed in Section 4.3).

3.2.2 Server Key and Fault Tolerance

The $ProFokus$ system has a single global public/private key pair, called the service key. This service key is used to encrypt/decrypt the keys stored by the servers and to digitally sign the responses sent back to the client. The private service key is stored at no single server. Instead, the key is split into a number of shares, each of which is stored at a single server and threshold cryptography [7] is employed to decrypt the keys and to construct signatures on the responses.

To sign a message in a distributed manner,

i. Each server creates a partial signature of the message using its share of the private service key
ii. Some server collects these partial signatures shares and combines them to produce the digital signature of the message

Similarly, to decrypt an encrypted key,

i. Each server creates a partial decryption of the encrypted key using its share of the private service key
ii. Some server collects these partial decryption shares and combines them to get the decrypted value of the encrypted key.

The $ProFokus$ system employs a $(n, t+1)$ threshold cryptography scheme, in which $t+1$ or more shares are needed to digitally sign a message or decrypt a value encrypted with the public service key. These shares are given to the servers through an initial dealing phase which is performed by an operator only once when the system is first deployed. By the Compromised Servers assumption (Section 3), a set of $(t+1)$ servers has to include one server that is correct and not compromised. So, each threshold cryptographic operation is performed only if some correct server has received sufficient evidence to justify executing the operation. This implies that to falsely generate a partial signature or to decrypt the keys stored in the $ProFokus$ servers, an adversary needs to compromise at least $(t+1)$ servers and this is ruled out because of our Compromised Servers assumption. This forms the basis of the $t-fault\ tolerance$ supported by the $ProFokus$ system.

3.3 Co-ordination of Server Replicas

The $ProFokus$ system employs replication techniques to ensure the availability of the services it provides. Specifically, the system employs an $opaque\ masking$ Byzantine quorum system [14] to store the service state. In such a system, $t-fault\ tolerance$ can be achieved by having a total of $5t$ servers with the set of quorums comprising all sets containing $\left\lceil \frac{2n+2t}{3} \right\rceil$ servers. The quorums of servers are defined so as to satisfy the following two properties:

- **Intersection:** Given any two quorums Q_1 and Q_2, $|Q_1 \cap Q_2| \geq t + |Q_1 \backslash Q_2|$
- **Availability:** At any point in time, a quorum consisting of all correct servers is always available.

Each of the $ProFokus$ operations are broken down into a number of steps and each step is performed by a quorum of servers. Different quorums of servers may be involved in different steps of the same operation. This does not create a problem since the Intersection property ensures that the result of all previous steps will be correctly available at enough servers to employ voting to get the correct updated state.

The detailed descriptions of the various protocols are given in the Appendix Section. In this section, the main ideas behind the design of the protocols are explained.

Timestamps: Client requests are processed by a quorum of servers but not necessarily by all correct servers and hence, different correct servers might process different Update requests. Consequently, different bindings for a given name N are stored by correct servers. Timestamps are used determine the last updated binding.

The role of Delegates: Clients do not know the server public keys and hence, a client making a request cannot authenticate messages from a server and cannot determine whether that request has been correctly handled. The solution is for some server to become a **delegate** for the incoming request. The delegate presides over the processing of a client request and, being a $ProFokus$ server, can authenticate server messages and assemble the needed partial signatures from other servers, to create the signed response. Since the delegate may itself be compromised and there will a maximum of t such

compromised delegates, a client needs to send the request to at least $t + 1$ servers to ensure that the request is handled by at least one correct delegate. Because the responses from the delegate are digitally signed using threshold signatures, clients are able to identify and reject corrupted responses from compromised delegates.

Figure 1 gives a high-level view of how COCA operates by depicting one of the $t + 1$ delegates and the quorum of servers, working with that delegate to handle a client request. The figure shows a client making its request by sending a signed message to $t + 1$ $ProFokus$ servers. Each server that receives this message assumes the role of

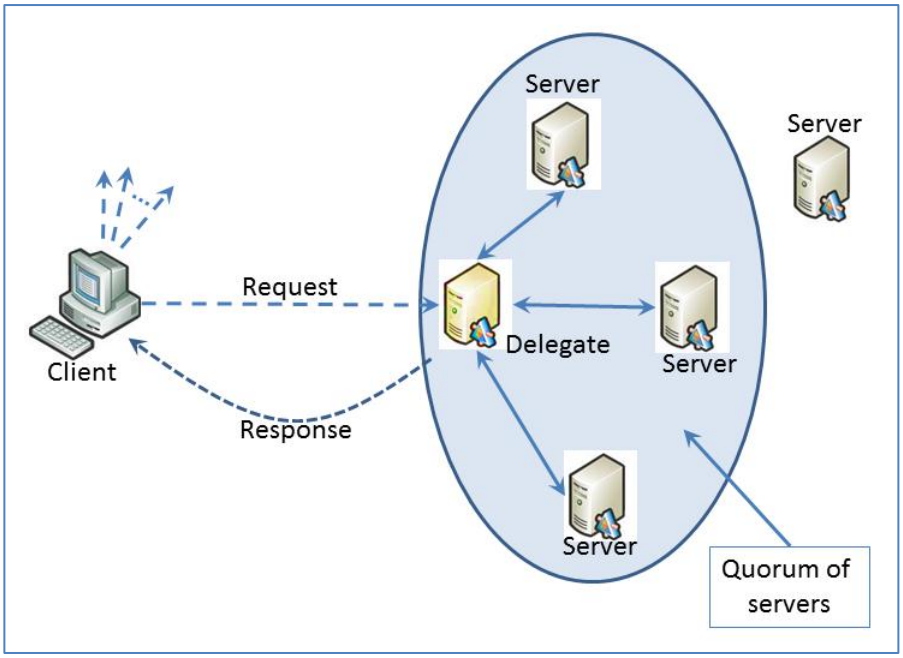


Figure 1 – Overview of client request processing

delegate for the request. A delegate engages a quorum of servers to handle the request (by sending that request to all $ProFokus$ servers) and constructs a response to the request based on the responses received from that quorum. The delegate then causes this response to be signed by the service; this involves running a threshold signature protocol in cooperation with t other servers. Once signed, the response is sent by the delegate to the

client. Upon receipt, the client checks that the response is correctly signed by the service and contains the client's original request; if no correct response has been received within a specified period of time, then the client simply again sends the original request to $t + 1$ servers.

Since the request is handled by $t + 1$ delegates, there is considerable duplication of effort by the servers and also, the same request will be handled multiple times. This does not pose a problem since the $ProFokus$ read operations are designed to be idempotent, while the $ProFokus$ Update operation is designed in such a way that only it is only executed once.

Self-Verifying Messages: Compromised delegates might attempt to produce an incorrect (but correctly signed) response to a client by sending erroneous messages to the other servers. The defense against this is to use the notion of self-verifying messages. A self-verifying message is comprised of:

- Information the sender intends to convey
- Evidence enabling the receiver to verify, without trusting the sender that the information being conveyed by the message is consistent with some given protocol and also is not a replay.

In the $ProFokus$ system, every message a delegate sends on behalf of a request contains a transcript of relevant messages previously sent and received in processing that request (including the original client request). Because messages contained in the transcript are digitally signed by their original senders, a compromised delegate cannot forge the transcript. And, because the members of the quorum participating in the

protocol are known to all, the receiver of such a self-verifying message can independently establish whether messages sent by a delegate are consistent with the protocol and the messages received.

Concurrency: There may be multiple clients which may need to read the key values for a particular name concurrently. This does not pose a problem since the read operations are not state-altering operations. However in cases of concurrent state-altering update operations for the same name, the $ProFokus$ system makes no guarantees about the outcome of such operations. The clients must have their own synchronization mechanisms to ensure that the update operations are not executed concurrently.

3.4 Protocols

This section gives a high-level description of the protocols that the $ProFokus$ system employs to provide the key storage functionality.

3.4.1 $Read\_timestamp$

This is the simplest of the three operations and is used the get the current timestamp for a name N. This request needs to be made before sending the update request, since the response for this request is then passed in the update request. This is done in order to make the update request idempotent, which we will see in Section 3.4.2.

Invocation message $Req_{TS}: < read\_timestamp, N >_c$

Confirmation Message $\Re_{TS}: < timestamp_{response}, N, \tau, Req_{TS} >_{ProFokus}$

Figure 2 – Client Protocol for *read_timestamp* operation

Figure 2 gives the invocation message and the confirmation message for this operation, carried out by client c to read the timestamp associated with a name N. The invocation message is digitally signed by the client (denoted by $<>_c$) and the confirmation message is signed by the $ProFokus$ system (denoted by $<>_{ProFokus}$). The third field τ in the confirmation message is the current timestamp value.

The following are the steps performed by a delegate D, who receives the invocation message to read a timestamp for a particular name N:

1) D forwards the request to all the $ProFokus$ servers and awaits timestamps for name N from a quorum of servers
2) D selects the timestamp τ returned by a majority of servers or the higher timestamp in case there is multiple servers sets with the same cardinality.
3) D generates a signature on the response by sending it to other servers, who send back partial signatures to D. D combines these partial signatures to get the signed response containing τ and sends this back to the client.

3.4.2 *Update*

This operation is used by the clients to create or update a name-key binding in the

Invocation message $Req_{update}: < update, \Re_{TS}, N, E(k), \pi(k, c) >_c$

Confirmation Message $\Re_{update}: < updated, N, Req_{update} >_{ProFokus}$

Figure 3 - Client Protocol for *update* operation

$ProFokus$ system. Figure 3 gives the invocation message and the confirmation message for this operation, carried out by client c to associate a key k with name N.

The key value k is encrypted using the service public key and is denoted by $E(k)$. This is done to ensure the confidentiality of the key value while in transit. The timestamp response message $\Re_{TS}$ is included so as to make the update operation idempotent. The inclusion of the whole message and not just the timestamp value is done since the client cannot be trusted to provide the correct value of timestamp and malicious clients might send a large value to exhaust the timestamp space. With the signed response, the servers can check the validity of the response message and be sure that the sent timestamp value is indeed valid.

The last field $\pi(k, c)$ is a non-malleable proof [11] that the client c knows the plaintext k, along with the ciphertext $E(k)$. This is required because in its absence, a malicious client could then read the key value k as follows: Intercept Req_{update}, copy $E(k)$ into a new $update$ request for some new name N' and subsequently perform a $read\_key$ operation for N'. The details about the working of the non-malleable proof are outside the scope of this thesis.

The following are the steps performed by a delegate D, who receives the update invocation message to associate a key k with name N. For each name, each server also keeps a hash of the last update request.

1) D forwards the request R to all the $ProFokus$ servers and awaits confirmation messages for that request from a quorum of servers.
2) On receiving such a forwarded message from D, each server updates its local database only if one of the following conditions hold:

a) the timestamp for name N in the client request is greater than or equal to the timestamp it stores for the same name.

b) the local hash stored for name N is equal to hash of the current request R

Each server sets its local timestamp to one more than the timestamp received in the request R and the local key-value as the one received in the request. This ensures that the condition a) is satisfied only once for a update request. Condition b) might be satisfied multiple times but each time the same update takes place. This ensures that the update request is idempotent.

3) D generates a signature on the response by sending it to other servers, who send back partial signatures to D. D combines these partial signatures to get the signed response containing τ and sends this back to the client as well as other servers.

3.4.3 *Read_key*

This operation is used by the clients to read the key value currently associated with a name. Figure 4 gives the invocation message and the confirmation message for this operation, carried out by client c to read the key value for a name N.

To ensure the confidentiality of the key value while in transit, the process of blinding, as discussed in Section 2.2.3 is employed. In the invocation message, the client sends a

Invocation message $Req_{key}: < read\_key, N, E(b_c), \pi(b_c, c) >_c$

Confirmation Message $\Re_{key} : < key\_response,\ N,\ k_N * b_c,\ Req_{key} >_{ProFokus}$

Figure 4 - Client Protocol for *read_key* operation

random blinding factor b_c in encrypted form. The blinding process works as follows:

For a homomorphic cryptosystem such as RSA,

$$E(k_N) \times E(b_p) = E(k_N \times b_p)$$

holds and, by design, each $ProFokus$ server stores $E(k_N)$. Thus, each $ProFokus$ server can compute $E(k_N \times b_p)$ and can then employ threshold decryption to obtain $k_N \times b_p$ without ever materializing k_N or b_p, so the confirmation message sent to c contains $k_N \times b_p$, which is undecipherable without b_p. And client c recovers k_N by dividing $k_N \times b_p$ by b_p.

The last field in the invocation message is $\pi(b_c, c)$, the non-malleable proof that the client c knows the plaintext b_c, along with the ciphertext $E(b_c)$. Without this proof, a malicious client c_m can determine the key value k as follows: Intercept $E(b_c)$. Intercept $\Re_{key}$ to learn $k_N * b_c$. Now the client can initiate a $read\_key$ operation for some name $N'$for which it is authorized to read, with the intercepted blinding factor $E(b_c)$ to obtain $b_c * k_{N'}$. Since the client is authorized to read N', it can get $k_{N'}$ by doing another read with some known blinding factor. Having known $k_{N'}$, the client can learn b_c and hence learn k_N. The proof of plaintext knowledge ensures that the $read\_key$ operation by a malicious client, with the intercepted blinding factor $E(b_c)$ is rejected.

The following are the steps performed by a delegate D, who receives the invocation message to read the key value for a particular name N:

1) D forwards the request to all the $ProFokus$ servers and awaits bindings for name N along from a quorum of servers
2) D selects a set λ of servers such that the set constitutes a majority of servers in some quorum and each server in that set has the same timestamp. If some servers are compromised and they decided to team up with the outdated servers, then there is a possibility that there might be two such sets existing with the same

number of servers, but with different timestamps. In that case, the set with the higher timestamp is selected, since that contains the most recently updated servers.

3) Along with the binding in step 1, each server also responds with a partial decryption of the key value, using its own key share. D combines the partial shares returned by the servers in set λ to obtain the required key value k.
4) D invokes the threshold signature protocol to sign the response containing k and sends that response back to the client

The Quorum Availability property of the $ProFokus$ system discussed in Section 2.1.1 ensures that a quorum of correct servers is always available, so the first step in each of the above protocols which requires waiting for responses from a quorum of servers, are guaranteed to terminate. Also, since only t out of the total servers can be compromised at any point, there are always at least $t + 1$ correct servers available and hence the compromised servers cannot prevent a delegate from using threshold cryptography in constructing the signature for a response. Thus, the threshold signature step in the above protocols cannot be disrupted by compromised servers.

3.4.4 Masking

As we have seen in the update protocol, the key k to be stored in the $ProFokus$ system is encrypted using the service public key. This implies that the maximum size of the keys ($|k_{max}|$) that can be stored is directly related to the maximum secret size ($|S_{max}|$) supported by the threshold cryptosystem used in the $ProFokus$ system. Also since the key value k to be read is blinded so as to ensure its confidentiality while in transit, the key size is also limited by

$$|k_{\max} * b| \leq |S_{max}|$$

In order to enable the storage of keys with greater length, the process of masking is introduced. This process to store a key value k for name N works as follows:

i. Generate a random key K_s using a symmetric key algorithm like AES [19]. The size of this symmetric key (Mask key) is dependent on the level of security required, but should be less than $|S_{max}|$.

ii. Associate a random name N_s with the symmetric key K_s and store this binding in the $ProFokus$ system using the update operation

iii. Store the key value k for name N using the update operation. However in the invocation message, instead of encrypting the key value with the public service key, encrypt the key value using the symmetric key K_s. This allows the size of the key-value to be independent of the threshold cryptosystem and hence allows the storage of longer keys. The corresponding key value stored in the $ProFokus$ system is called the $masked\ key\ value$.

To read the key value k associated with a name N, a new operation called as $read\_masked\_key$ is introduced. The working of this operation is fundamentally the same as the $read\_timestamp$ message; however the masked key value is returned instead of the timestamp. The details of this protocol appear as part of the Appendix. The process of reading the key value now works as follows:

i. Invoke a $read\_key$ operation for name N_s to get the symmetric key K_s.

ii. Invoke a $read\_masked\_key$ operation for name N to fetch the *masked key value,*k_m

iv. Decrypt the masked key value k_m using the symmetric key K_s to obtain the required key value k.

CHAPTER FOUR - PROACTIVE RECOVERY

The $ProFokus$ system described in this thesis has been designed to be Byzantine fault-tolerant through the use of Byzantine quorum systems [14]. This has been coupled with threshold cryptography [8] to ensure the confidentiality of the data stored in the system. These methods are effective and function correctly as long as the assumption of the limited number of failures during the lifetime of the system holds. However, many systems are designed with a goal to operate for a long time (long-lived systems [18]), making this assumption problematic. Also, threshold cryptography alone does not suffice to defend against $mobile\ adversaries$ [16] which attack, compromise and control a server for a limited time, and then move onto another. Given enough time, a mobile adversary can compromise $t + 1$ servers, learn the $t + 1$ shares of the service private key and thus reconstruct key and decrypt the data stored at the servers.

To defend against this, a proactive recovery subsystem has been developed that

- Recovers and restarts all the servers from a secure code base to remove any vulnerability introduced by an adversary. Since a server that has experienced a Byzantine failure may appear to behave correctly even when compromised, all servers including correct ones must be recovered.
- Refreshes the key shares of the threshold cryptosystem at the servers such that the new and old shares are independent and an adversary cannot combine old shares with new shares to reconstruct the private key.

This chapter discussed the proactive resilience model that is implemented in the $ProFokus$ system.

4.1 Proactive Resilience Model

The proactive recovery mechanism can be used to rejuvenate and recover the components of the system, as long as the mechanism has timeliness gaurantees and can be executed in a synchronous environment. The Proactive Resilience model [25] is proposed in this regard and consists of a hybrid distributed model composed of two parts: the proactive recovery subsystem and the payload system, the latter being proactively rejuvenated by the former. Each of these parts have different synchrony assumptions and have different fault models. The payload system is the system that executes the normal applications like the $ProFokus$ system and as we have seen in Section 3, operates in an asynchronous Byzantine environment. The proactive recovery subsystem, however is deployed in a secure distributed component called the Proactive Recovery Wormhole (PRW) and operates in a synchronous environment. The fault model of this subsystem

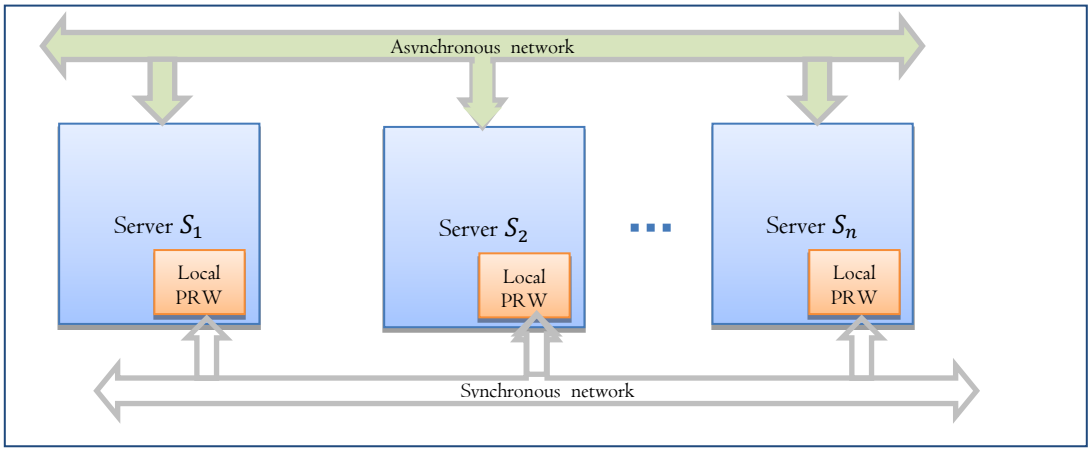


Figure 5 – The architecture of a system with a PRW

depends upon the functions that the rejuvenation procedure needs to perform, which itself is dependent upon the payload system that needs to be proactively recovered.

The architecture of a system with a PRW is shown in Figure 5. There is a local PRW per payload system and all the local $PRWs$ are connected through a synchronous and

secure control channel, which is isolated from other networks. The set of all local $PRWs$ is called $the\ PRW\ Cluster$ or simply $the\ PRW$. The local $PRWs$ need to be seperated from the payload systems. This seperation can be achieved in two ways:

- Physical: Implement local PRW in a separate tamper-proof hardware module
- Logical: Use software virtualization [3] to implement the payload and the proactive recovery subsystem on the same hardware.

The implementation of this seperation is outside the scope of this thesis.

4.2 Assumptions

To integrate the Proactive Recovery Model with the $ProFokus$ system, some additional assumptions need to be made:

- **Fault Model:** In order to enable the PRWs to recover the payload state from other PRWs during the rejuvenation procedure, it is necessary that the local $PRWs$ function correctly despite the compromise of the payload applications. In case any of the $PRWs$ fail, then the other $PRWs$ may not be able to recover the correct state.
- **Periodic Timely Rejuvenation Service:** We assume the presence of a periodic timely rejuvenation service that can be used by the $PRWs$ to execute periodic rejuvenations. Specifically, the PRW cluster executes a rejuvenation procedure F in rounds and each round is triggered within T_P from the last triggering. Any local PRW can trigger F. All other $PRWs$ then start executing F within time T_π. Also, once all PRWs start executing F, then the execution is bounded by T_D. Thus the worst case execution time for F in each round is $T_\pi + T_D$.

- **Key Pairs:** Each PRW has a public-private key pair, with the public key known to all other $PRWs$. These key pairs are used to enable authenticated comunications between the $PRWs$.
- **Server Code:** Each local PRW has a copy of the server code that is used to restart the payload server.

4.3 Rejuvenation Procedure:

The rejuvenation procedure F to be executed by the PRW consists of following stages:

1. **State Recovery**: This stage involves recovering the bindings that each $ProFocus$ server stores.
2. **Threshold Share Recovery**: This stage involves recovering the partial share of the private service key that each server stores.
3. **Threshold Share Refresh** : This stage involves refreshing the partial shares and generating a new sharing of the private service key
4. **Payload Restart**: This stage involves restarting the payload applications with a clean and correct state.

Window of Vulnerability: The time for the rejuvenation procedure T_D is bounded by three variables: T_{SR} – time for state recovery, T_{TSR}- time to recover and renew the partial share and $T_R -$ time to restart the payload application. The execution time of F is bounded by T_D. If the time T_D elapses before the rejuvenation procedure could be completed, the local PRW can contact an administrator that can take corrective actions.

The execution of the $ProFokus$ system is divided into periods called "windows of vulnerability". A window of vulnerability starts when the system starts accepting client requests with all the payload servers in a correct and updated state and it ends when a local PRW triggers the proactive rejuvenation procedure. After the rejuvenation procedure ends, the system enters the next window of vulnerability. Figure 6 shows the relation between the window of vulnerability and the execution of the periodic rejuvenation procedure. The system is unavailable during the execution of the rejuvenation procedure and we try to minimize this duration.

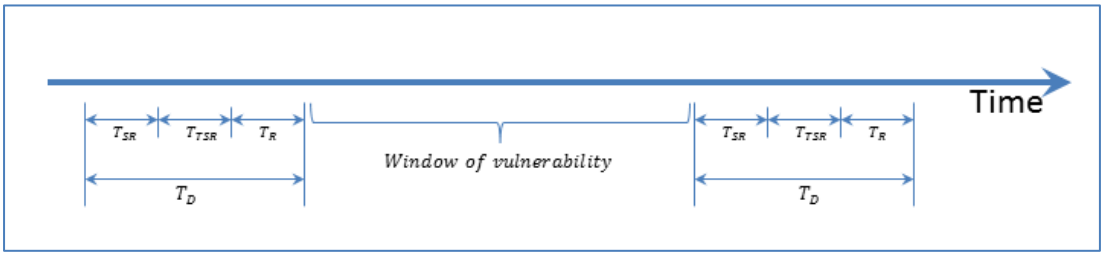


Figure 6 – Relation between window of vulnerability and Proactive Recovery

Before the rejuvenation procedure is started, each local PRW instructs the payload server to dump its current state. Note that compromised payload servers may not follow this procedure correctly and may dump an incorrect state. This does not create a problem since the $ProFokus$ system is designed to tolerate up to t failures and hence the correct state can be recovered even if some servers are compromised. After the state has been successfully dumped in case of correct payloads or after the certain timeout in case of compromised payloads, each local PRW will shut down the payload servers and start the rejuvenation procedure.

4.3.1 State Recovery

The state of a $ProFokus$ server consists of a set of bindings of the form

$< timestamp, name, key- >$ or $\beta: < \tau_N, N, E(k_N) >$

The $ProFokus$ system is modeled as a $masking$ byzantine quorum system as shown in Section 3.1 and at any point in time, for each binding β, the number of servers with the updated state is greater than the number of out-of-date servers and the number of compromised servers combined. If $\chi_{correct}$ denotes the correct updated state, χ_{old} denotes an old state and $\chi_{compromised}$ denotes a compromised state, then

$$|\chi_{correct}| > |\chi_{old}| + |\chi_{compromised}|$$

is true for each binding. During the state recovery stage, each PRW sends to every other PRW the state χ, dumped by its payload server. The global correct state consists of all names with the same $key - value$ and $timestamp$ and included in a majority number of states. Since a compromised payload might dump a state with an unbounded size, there needs to be a bound on the size of each accepted dumped state χ. Any state larger than this bound is automatically rejected by its local PRW.

At the end of this stage, each local PRW would have constructed the same global state χ_{global} and this state would contain the updated key-value and timestamp for each name.

4.3.2 Threshold Share Recovery

During the initial secret sharing done by the trusted dealer, the private service key was shared among the payloads using a $(n, t + 1)$ threshold signature scheme [20]. However, since the payload servers are subject to Byzantine failures, these key shares can be deleted or modified, in case the payload gets compromised by an adversary. To detect the modification or to recover the lost share, each share is also stored as a backup in the local PRW, when the initial dealing is done. Note that the backup shares are stored

locally by each local PRW and not accessible to the payload and hence not accessible to the adversary as well. These backup shares are used to do the share renewal in the next stage.

4.3.3 Threshold Share Refresh

A mobile adversary might compromise $t+1$ payloads given sufficient time and hence might collect $t+1$ shares of the service private key. To prevent this, a proactive share refreshing protocol is run in this stage, which generates a new set of shares and deletes the old set. The exact mechanism depends upon the threshold scheme used to do the initial sharing. The RSA threshold signature scheme [21], for example uses Shamir secret sharing approach [20] to distribute the key shares among the participants. Consequently, Herzberg's share renewal scheme [10] can be employed to refresh the shares. In this scheme, each share can be renewed by having each local PRW generate shares of "0" using the same $(t+1, n)$ secret-sharing used in the initial dealing phase and then distributing it to other $PRWs$. Upon receiving a share of "0", each local PRW can add it to its current share of the private service key, which would also have been generated using a $(t+1, n)$ secret-sharing. The combination of the shares will still yield the correct private key due to the linearity in polynomial addition.

4.3.4 Payload Restart

In this stage, the payload application code is reloaded from the local $PRWs$ and the payload server is rebooted. The payload server is restarted with the recovered correct state obtained in the first stage and with the new partial shares in the third stage. Once the

payload servers have successfully rebooted, the systems enters the next window of vulnerability and the $ProFokus$ system is ready to be used by the clients.

CHAPTER FIVE - INTEGRATION WITH GRIDSTAT

5.1 GridStat

GridStat is a communication technology, being developed to provide real-time and rate-based data delivery services to the electric power grid, but it has many other applicable domains where the data to be distributed is periodic in nature. It belongs in a class of software called middleware, which functions between the operating system and the application software. GridStat facilitates the communications between the control system devices and adds additional capabilities to improve the quality of service (QoS) of inter-device communications. GridStat handles network resource allocations, redundant communications pathways, secure links and protocol translations in a way that is transparent to the control system devices [6].

Compared to today's control system technology, GridStat's flexibility allows better performance, robustness, and security. GridStat lets data be delivered flexibly in a peer-to-peer fashion without requiring a centralized collection point, thus reducing latency and eliminating a single point of failure. GridStat has a hierarchical management infrastructure that maps onto the natural hierarchy in the grid. This enables utilities and other participants in the power grid to configure their own security policies to control access and resource usage to meet their own security requirements. Architecturally, GridStat is an overlay network based on the publisher-subscriber paradigm which is a communication paradigm that supports dynamic, asynchronous, many-to-many communications in a distributed environment and couples this with hierarchical QOS

management to allow the dissemination of periodic status updates to the interested subscribers [2].

GridStat components are organized into the Data Plane (DP), Management Plane (MP) and the Security Management Plane (SMP). Each of them addresses specific functionality to support timely and efficient information exchange. The following sections provide an overview of these planes and their functionalities.

5.1.1 Data Plane

The data plane consists of the publishers, subscribers and the forwarding engines (FEs) called status routers (SRs) that route the status updates from the publishers to the subcribers. Given the relatively static and completely planned power grid environment, static routing is employed and the SRs merely forward the packets without looking at their content and do not have to update their routing tables often.

Apart from forwarding the SRs are also responsible for rate filtering and multicast. Subscribers may request for different rate of updates and rate filtering facilitates this. Multicast is also efficiently supported by never repeating an update on a link. To ensure that updates reach the publishers, they are sent over multiple different paths thus employing redundancy to improve availability [1].

Figure 7 shows the architecture of GridStat depicting the data plane and management plane components.

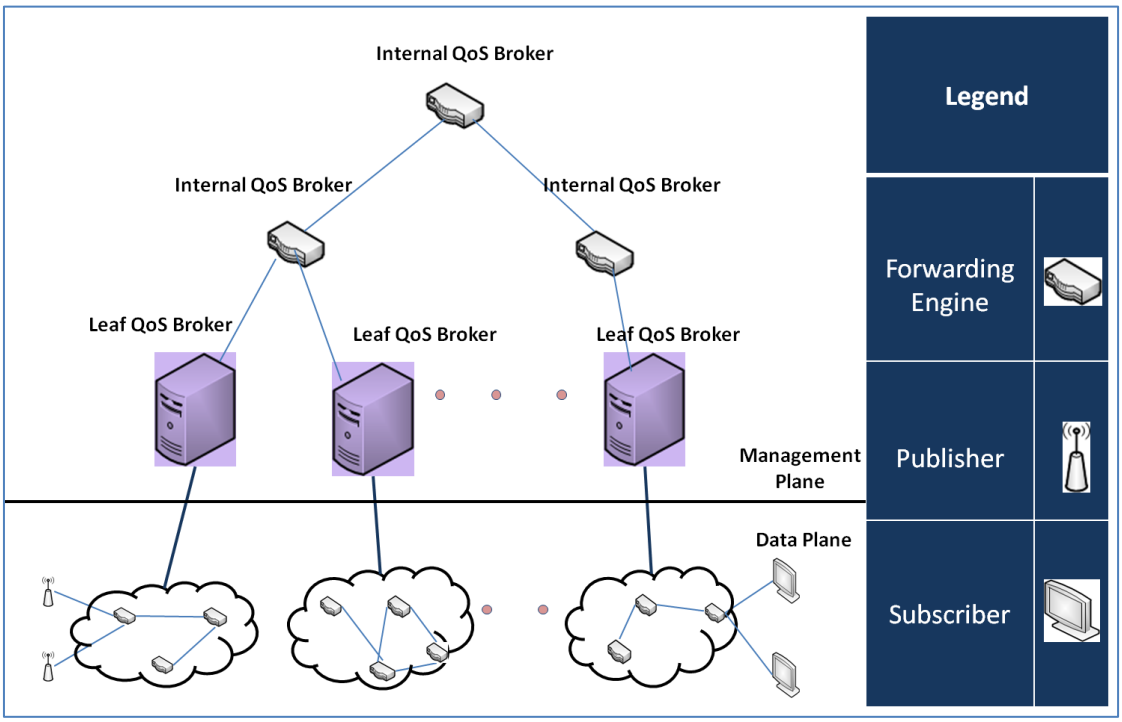


Figure 7 - Architecture of GridStat

5.1.2 Management Plane

The Management plane consists of quality of service brokers (QoS brokers) that are responsible for control decisions and managing the resources of the data plane. Among their most important responsibilities are granting or denying subscription and publication requests, path allocation which involves configuring the SRs' forwarding tables and ensuring that QoS requirements are met.

The QoS brokers are organized in a hierarchy that maps onto the natural hierarchy in the electric power grid. The leaf nodes in the QoS broker hierarchy are the leaf QoS brokers. Leaf QoS brokers have jurisdiction over a cloud of data plane nodes. The leaf QoS brokers interact with other interior QoS brokers to aid decision making in case of inter-cloud publications/subscriptions.

When publishers need to publish a variable, they contact their own leaf QoS broker that verifies if the publication can be supported. If the publication can be supported the

Leaf QoS broker registers it. Similarly, subscribers contact their leaf QoS brokers for subscription requests and the leaf QoS broker grants or rejects a request according to subscriber permissions and QoS requirements.

5.1.3 Security Management Plane

The security architecture of GridStat is built upon the idea of using transparent interchangeable software security modules to achieve the needed confidentiality, integrity, and authentication. This is accomplished by developing a security extension to the management plane, called the security management plane (SMP), that generates keys and assigns sets of modules from a module repository, to the publishers and subscribers, on a per publication granularity according to dynamic policies. Assigning sets of modules and keys on a per-variable granularity enables the security system to address the different needs of different multicast streams with status updates, which are called publications. For some publications the need for confidentiality might be the strongest concern, and thus be assigned strong encryption modules, others might put emphasis on integrity and obfuscation, while others again might have strict real-time requirements and only use weaker, but faster, security modules.

To mitigate both the scaling and the single point of failure problem, the SMP is organized in a hierarchy, which mimics the hierarchy of the management plane, with special type of servers called leaf Security Management Servers (leaf-SMS) as leafs and interior security management servers (interior-SMS) as internal nodes. The leaf-SMSs serve a single GridStat cloud by assigning keys and modules to all publications within that cloud. They also provide the publications with access control by controlling the access to their assigned modules and keys. Without the correct modules and keys a

subscriber will not be able to access the information of the publication, thus the leaf-SMS controls access to all information published in the cloud it services.

The interior-SMSs do not contain any information about publications; their only job is to forward inter-cloud subscription requests to the leaf-SMS that controls the requested publication in a DNS like fashion [27]. They are needed in order provide the SMP with a level of flexibility that a peer-to-peer scheme of direct communication between leaf-SMS cannot support.

A publication or subscription request is treated in two phases, first the data plane node, a publisher or subscriber, fills out a policy file with the information it has access to such as the name of the publication, then it sends this policy to the SMP. The SMP then completes the policy with module and key assignments if the data plane node has the needed credentials, and sends a copy of the completed policy back. The data plane node then checks whether it has all the assigned modules in its own library of modules; if not, it requests the missing modules from the SMP.

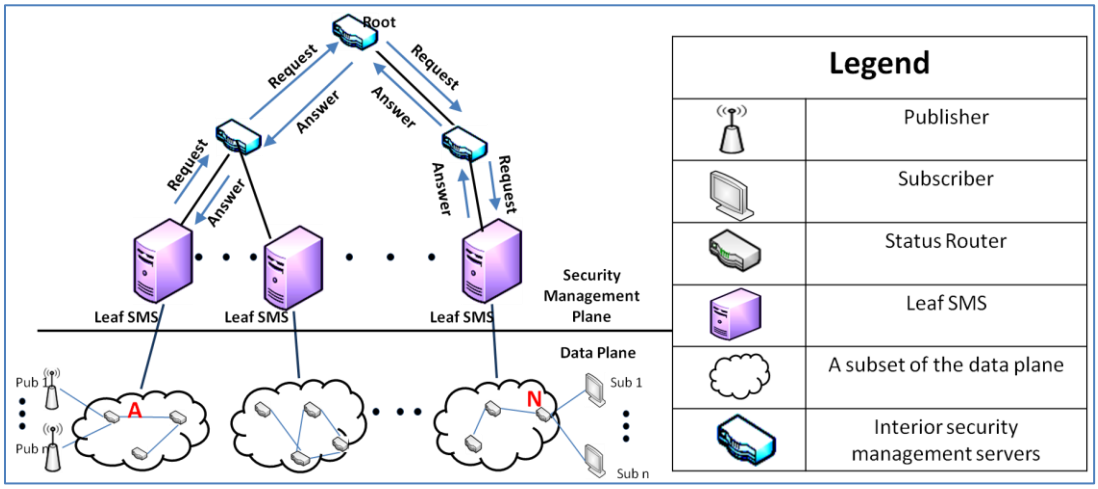


Figure 8 – Organization of Security Management Plane in GridStat

Figure above shows how the SMP is organized on top of the data plane. If Pub-1 wants to publish a variable it simply goes through the two round communication with the

leaf-SMS that controls cloud A, retrieving a publication policy and its associated modules. To set up an inter-cloud subscription, such as Sub-1 in cloud N subscribing to the new Pub-1 publication, the leaf-SMS that serves cloud N, and thus receives the first round of the subscription request from Sub-1, needs to forward the uncompleted subscription policy to the leaf-SMS that owns the publication policy in question, in this case the leaf-SMS controlling cloud A. This is done by forwarding the subscription policy to the parent SMS. The parent SMS checks whether the publisher, in this case Pub-1, is within the scope of any of its children with a hash table lookup. If the lookup returns false the policy is forwarded further up the hierarchy, but if the publisher is found within its scope it forwards the policy down to the towards the leaf-SMS that serves that publisher.

When the subscription policy reaches the leaf-SMS that controls access to the publication the leaf-SMS goes through the same process as with local subscriptions, it tests the subscriptions' credentials and if accepted fills out the subscription policy with module and key assignment before returning it, in this case through the hierarchy and back to the subscriber Sub-1. The subscriber then requests any missing modules that it needs based upon the received policy.

Leaf Security Management Server (Leaf-SMS)

The leaf-SMS is responsible for serving a single GridStat cloud, providing data plane security for that section of the data plane. It issues publication policies, with module and key assignments, to all publications that are registered in its cloud and issues subscription policies to all subscriptions to these publications, whether they are local or global.

Figure 9 depicts the logical structure of the leaf-SMS. At its core it has four databases keeping its state. The Publication/Subscription policy DB contains all the currently used policies for publications in the leaf-SMS's subset of the data plane and their subscriptions, the Security Group Policy DB keeps the SGPs that are defined for the leaf-SMS, while the Security Management Communication Policy DB (SMCP-DB) contains the Security Management Communication Policies (SMCPs) that specifies the security for the virtual point-to-point security management communication links. The fourth database is a module repository that contains the modules that are used to provide

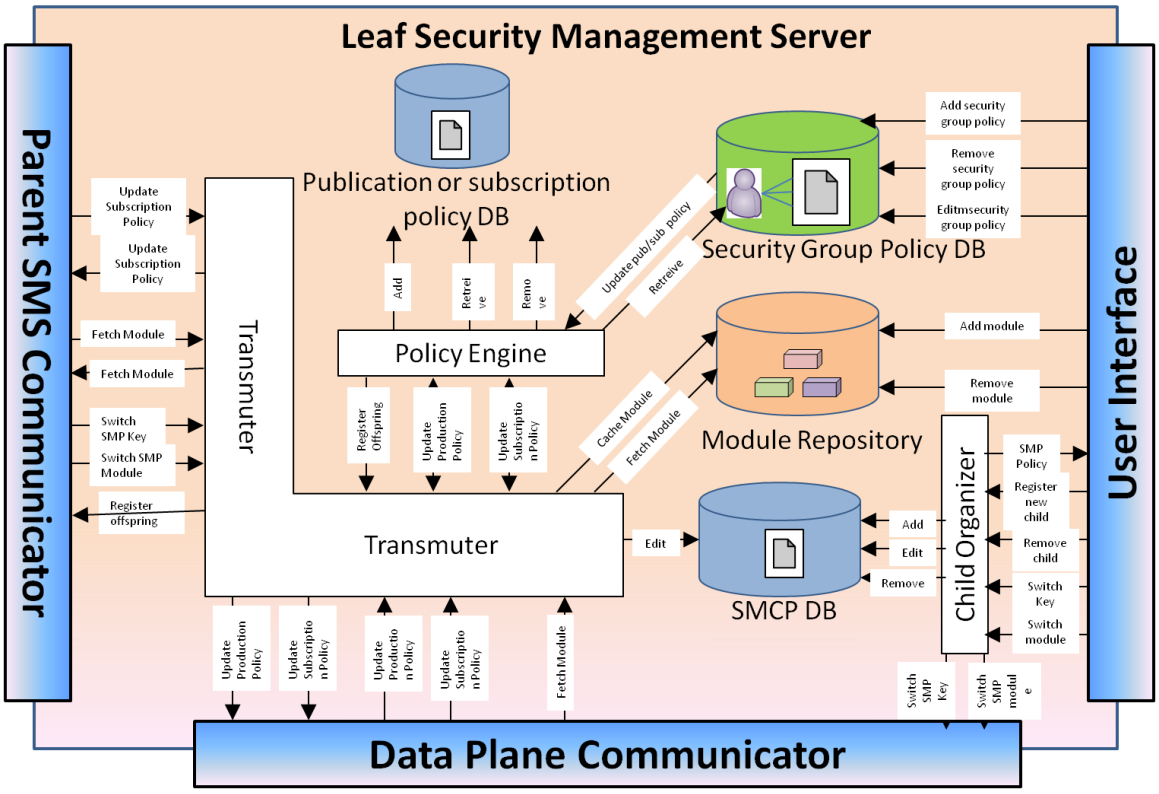


Figure 9 – Internal Structure of Leaf Security Management Server in GridStat

security for both the data plane and the security management plane.

A leaf-SMS has three interfaces as depicted in Figure 9. It has a user interface used by the operators to control the various functions. Secondly, it has an interface to its parent

interior-SMS called a Parent SMS Communicator and an interface to the data plane called the Data Plane Communicator, whose task is to handle the two-way communication with the publishers and subscribers in the leaf-SMS' cloud. Both inbound and outbound messages have to go through the Transmuter that looks up what modules that are currently assigned to communication with the source or target in the SMCP-DB, and applies these modules to the request before the command is passed on.

The Policy Engine is the main logical component in the leaf-SMS and controls the access to existing policies and the creation of new ones. Whenever it receives a policy for update from the data plane, it synchronizes it with the current policy for that publication or subscription in the Publication/Subscription Policy DB. When it receives a new publication policy, it retrieves the SGP for the security group that the publication policy specifies it wants to be a part of, and completes the policy with the information specified there, such as to set an expiration date on the policy and assigned modules. Before adding the new module to the policy database and returning a copy to the publisher, the Policy Engine generates keys to the assigned modules with the key generator specified in the module policy for each module. This can either be the default generator provided by the leaf-SMS itself, or a specially supplied key generator module for that security module.

5.2 GridStat Extension

The interchangeable security modules architecture of GridStat currently allows the system to support the confidentiality and integrity of the publications that flow through the data plane, as well as support authenticated communication both within the SMP and between the SMP and the data plane.

However, the confidentiality of the keys used for the publications is still a matter a concern. The current architecture although being highly modular, has the consequence of key leakage to the security management plane. Also, since the leaf-SMS has complete control and access to all the information in the cloud it controls, it becomes a highly alluring target for the attackers. The most conspicuous effect of a compromise of a leaf-SMS is the leakage of the publication keys of the publications in its cloud. However there are other consequences as well. As we have seen earlier, the subscribers issue a request for a subscription policy to be completed, to their leaf-SMS, that eventually contacts the leaf-SMS that controls the corresponding publication. This leaf-SMS fills in the keys and module information and sends the completed subscription policy back to original leaf-SMS and eventually back to the subscriber.

There are three types of communication taking place here:

- Subscriber to Leaf-SMS
- Leaf-SMS to Interior-SMS/ Interior-SMS to Leaf-SMS
- Interior-SMS to Interior-SMS

There are two Leaf-SMSs and a number of other interior-SMSs involved in the entire subscription process. The above communications have been secured through the use of preloaded shared keys provided to them by the operator out-of-band [4]. Hence, it is unlikely for an adversary to gain access to the subscription policies just by attacking the communication channels. However, the subscription policies and hence the publication keys will be available to the two leaf-SMSs and all the interior-SMSs that participated in the subscription process. A compromise of any of these entities will lead to the leakage of

the publication keys and hence there is a lot of effort required by the administrators to provide the same level of security for all the entities in the SMP.

The *ProFokus* system discussed in this thesis can be used to overcome this problem of key leakage. The first and foremost problem that needs to be tackled is the generation of the publication keys. In order to prevent the key leakage to the leaf-SMS, the responsibility of the key generation needs to be shifted from the leaf-SMS to the publisher itself. This will ensure that the keys are kept safe during the publication request phase. The next problem is the possibility of the leakage of keys during the subscription phase. This can be avoided by making the publisher store the keys in the *ProFokus* system and making the subscribers request the keys from the *ProFokus* system. These two approaches ensure that the data plane nodes, i.e. the original publisher and the authorized subscribers are the only entities that have knowledge of the keys.

To support the use of the described *ProFokus* system, the Data Plane Nodes need to be extended and some policies need to be changed. The changes needed in the Security Management Plane are minimal.

5.2.1 Data Plane Extension

The data plane consists of the publishers, subscribers and the status routers (SRs) that route the status updates from the publishers to the subscribers. To support the use of the *ProFokus* system, the publishers and subscribers need to be extended with new functionality.

Currently, the publishers and subscribers rely on the shared preloaded keys to communicate securely with the management plane and the SMP. However, this does not suffice for the communication with the *ProFokus* servers since the messages need to be

digitally signed. So, a public-private key pair needs to be generated for these nodes and the corresponding public key needs to be sent to all the $ProFokus$ servers. The private key is kept securely at the nodes and is used to digitally sign the requests. A new component called the Key Server Helper is added at both these nodes, that performs the following tasks:

- Loads the public service key of the $ProFokus$ system, so that it can encrypt the keys to be stored using the same.
- Loads the private key of the nodes, so that it can digitally sign the request message
- Creates the digitally signed request message in the appropriate format, according to the type of the request

Also, to ease the burden on the data plane nodes, the communication with the $ProFokus$ servers is channeled through the leaf-SMS. This avoids the need for configuring the nodes with information about the key servers. The leaf-SMS servers only act as request forwarders and the information in the request message is kept confidential by using encryption in case of update messages and blinding in case of read messages. The messages sent to the leaf-SMS are specific to the type of node and are discussed in the next two sections.

Publisher Extension

A publisher in GridStat is a producer of periodic status updates. Publishers register publications with the management plane, together with their rate and type, and get assigned a publication ID before starting to push status updates of that type with the specified rate into the data plane.

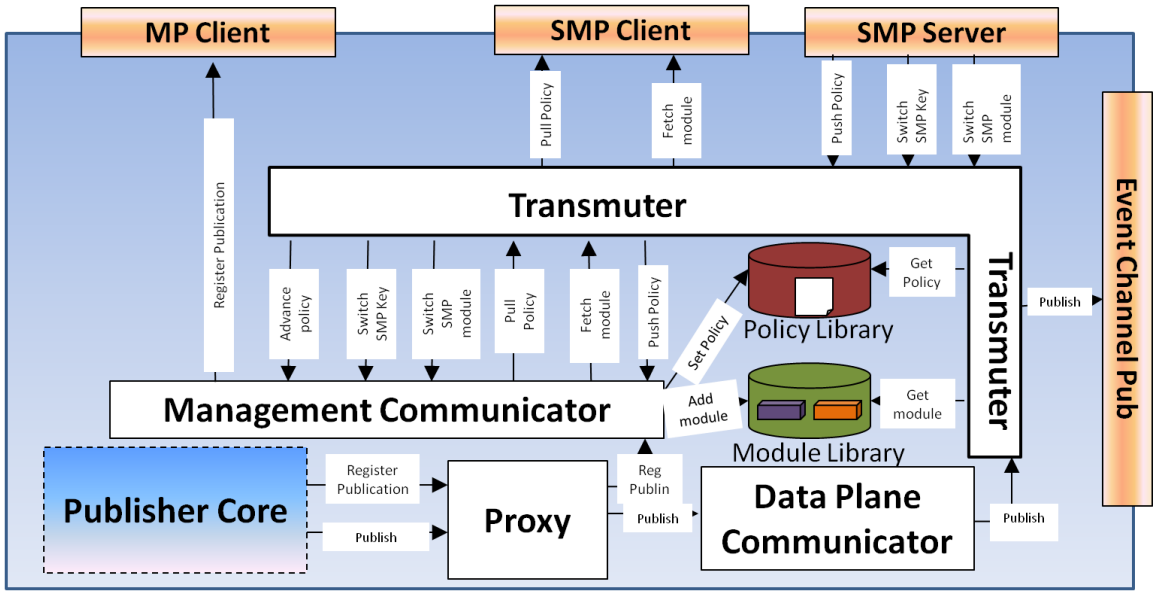


Figure 11 - Internal Structure of a Publisher in GridStat

The design of the original Publisher is shown in Figure 11. The Management Communicator is the component responsible for communications with the Management plane and the Security Management Plane. All communication with the security management plane goes through the Transmuter which applies a security module to

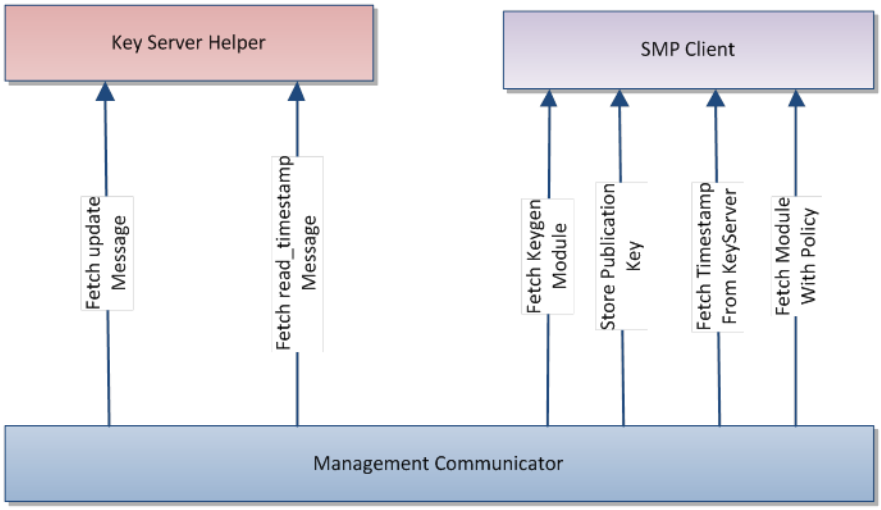


Figure 10 - – Extension of Management Communicator for a Publisher

secure the communication. The Module Library stores all the downloaded module files while the Policy Library holds the active publication policy for each of the publisher's publications. To support the use of the *ProFokus* system, there are four new messages

sent from the Management Communicator to the SMP Client, that is the entity that communicates with the leaf-SMS and these messages are shown in Figure 10.

The flowchart in Figure 12 shows the changes required in the process of the

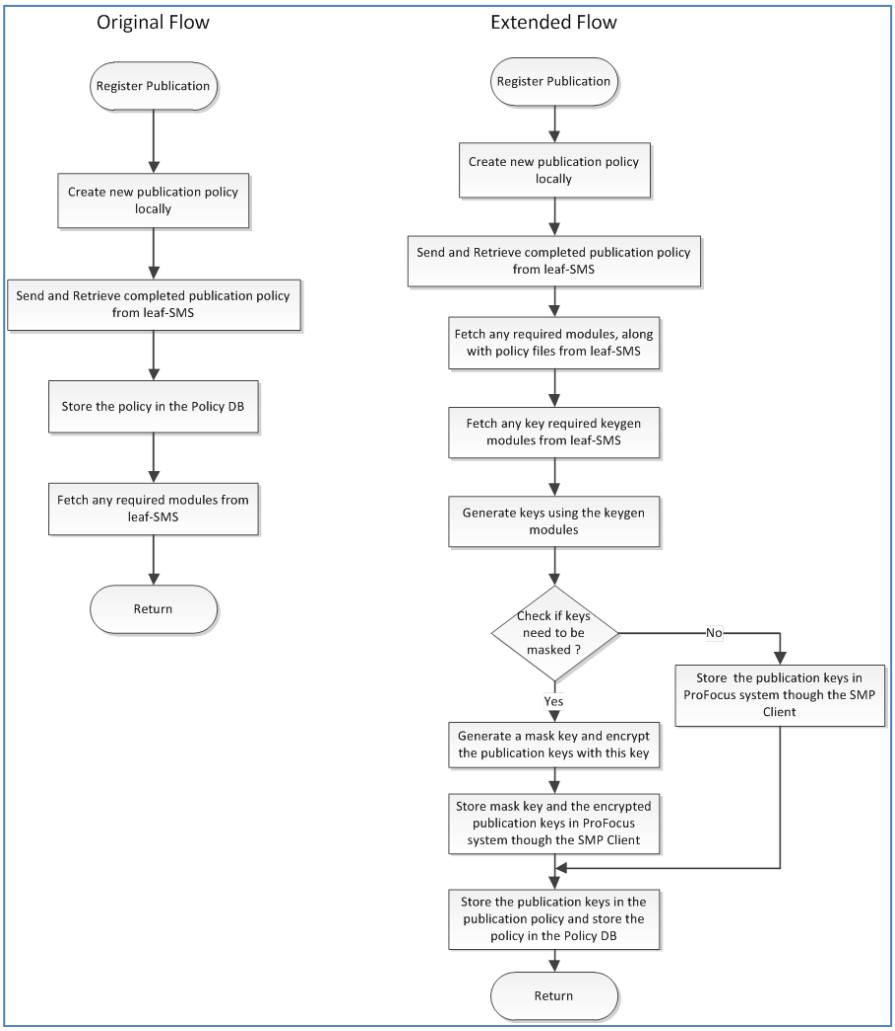


Figure 12 - Registration process for new Publication

registration of a new publication variable.

Subscriber Extension

Subscribers are consumers of the rate-based status updates published by the Publishers. A subscriber requests a subscription, with QoS requirements, to a published

variable from the management plane and if accepted, the subscriber will start receiving status updates with the specified QoS guarantees from the Publisher.

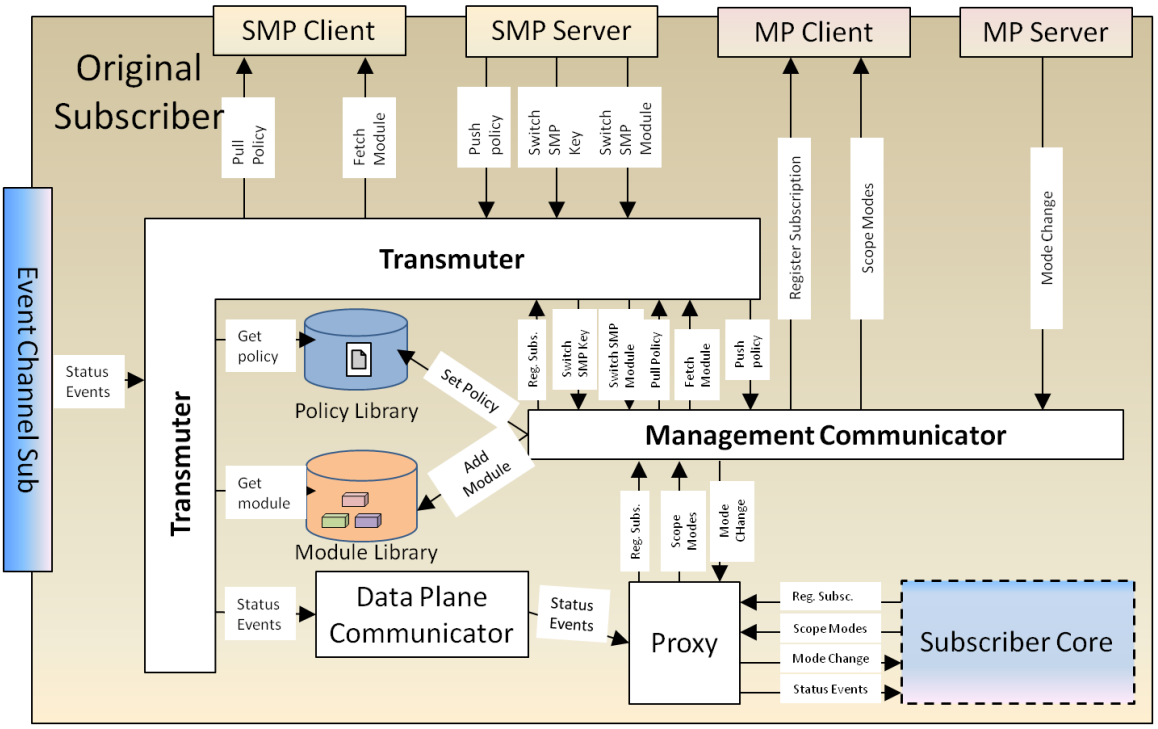


Figure 13 – Internal Structure of a Subscriber in GridStat

The design of the original Subscriber is shown in Figure 13. As with the original publisher, the Management Communicator is the component responsible for

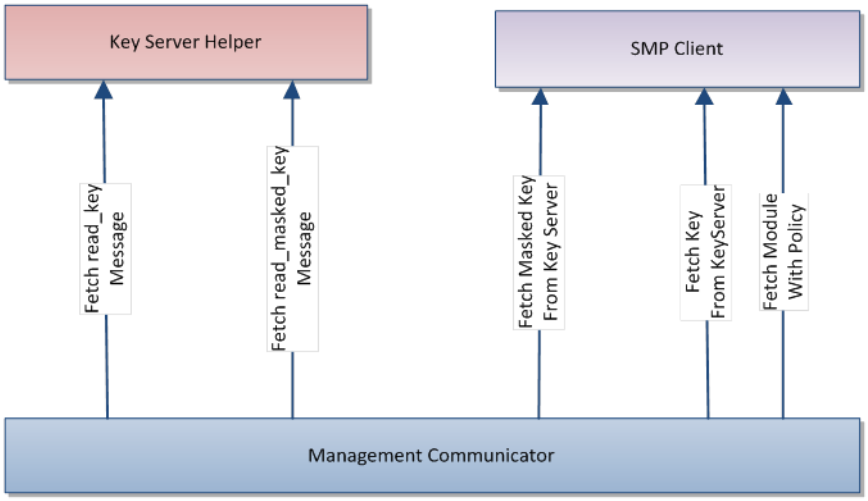


Figure 14 - Extension of Management Communicator for a Subscriber

communications with the Management plane and the Security Management Plane. All communication with the security management plane goes through the Transmuter which applies a security module to secure the communication. The Module Library stores all the downloaded module files while the Policy Library holds the active subscription policy for each of the subscriber's subscription. To support the use of the *ProFokus* system, three new messages are sent from the Management Communicator to the SMP Client and these messages are shown in Figure 14. The flowchart in Figure 15 shows the changes required in the process of the registration of a new subscription variable.

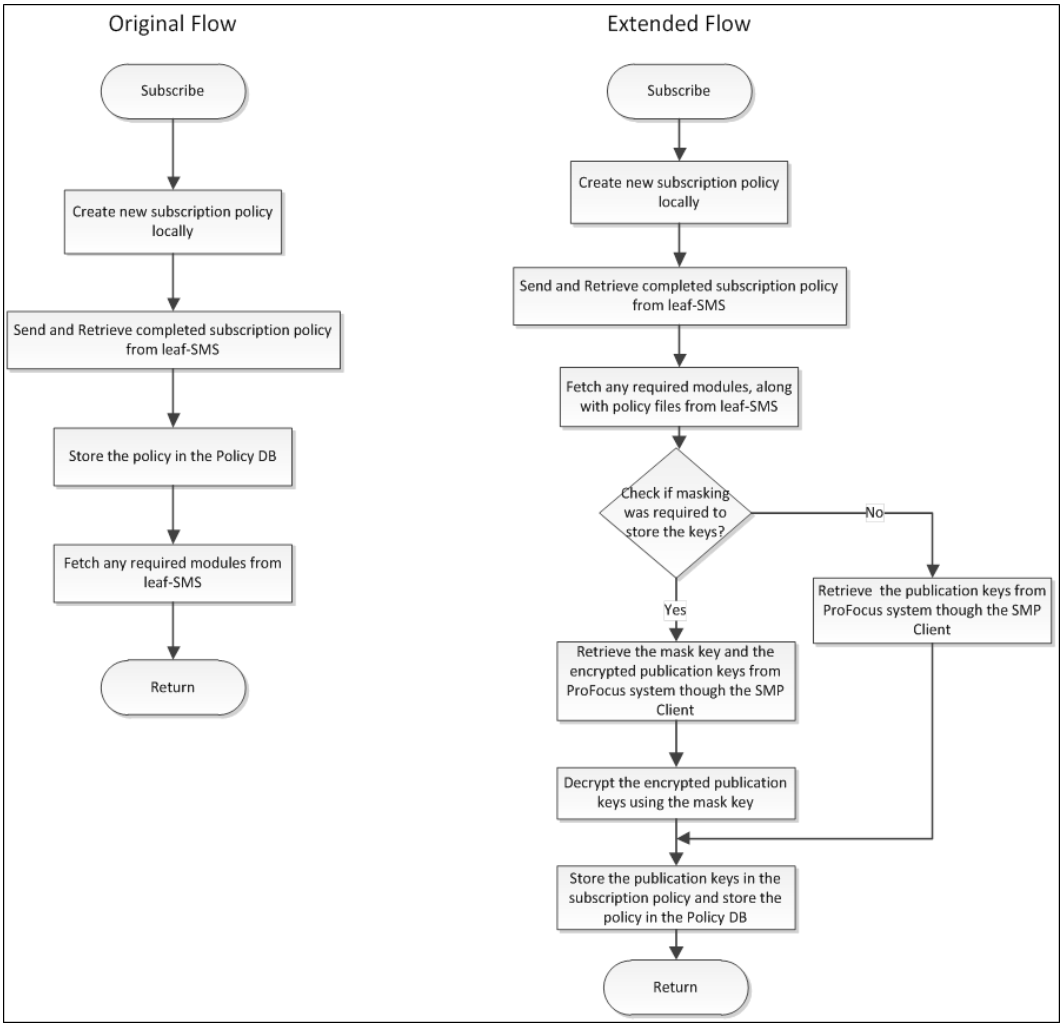


Figure 15 - Registration Process for new Subscription

5.2.2 Security Management Plane Extension

To support the use of the *ProFokus* system, there are some changes required in the leaf-SMS server. The interior-SMS remain unaffected.

Leaf-SMS Extension

As shown in Figure 10 and Figure 14, there are some new messages sent from the data plane nodes to the leaf-SMS and the leaf-SMS needs to be extended to handle these messages. Also, since the data plane nodes depend on the leaf-SMS for their communication with *ProFokus*, a new component called the "Key Server Communicator" is introduced in the extended leaf-SMS to handle the same.

Also, the responsibility of the generation of publication keys has been moved from the Policy Engine in the leaf-SMS to the data plane nodes. So, during the publication process, the leaf-SMS no longer generates the keys but instead it adds placeholders in the publication policy. The publisher uses these placeholders in combination with the publication ID as the names for storing the keys in the *ProFokus* system. During the subscription process, the same placeholders are added to the subscription policy and the subscribers retrieve the keys from the *ProFokus* system using these placeholders as the names.

Figure 16 shows the publication policy returned to the publisher by the leaf-SMS when the latter receives a publication registration request from the former. In this, the placeholder $0 is stored instead of the key. The Policy Engine at the Publisher generates the key and inserts it into the publication policy, shown in Figure 17. Also, the publisher stores this key in the key-service by sending corresponding requests to the leaf-SMS.

This new policy is exactly the same policy that would have been returned by the leaf-SMS in the absence of the *ProFokus* service integration.

```
<GridStatPolicy>
  <Type>Publication</Type>
  <PolicyNum>1</PolicyNum>
  <Publication>test_int_10</Publication>
  <PublicationId>-817033956</PublicationId>
  <Publisher>gridsim.stn0</Publisher>
  <SecurityLevel>1</SecurityLevel>
  <Modules>
    <Module Type="Encrypt">
      <Name>AES</Name>
      <Key>$0</Key>
    </Module>
  </Modules>
</GridStatPolicy>
```

Figure 16 - Publication Policy with placeholder

```
<GridStatPolicy>
  <Type>Publication</Type>
  <PolicyNum>1</PolicyNum>
  <Publication>test_int_10</Publication>
  <PublicationId>-817033956</PublicationId>
  <Publisher>gridsim.stn0</Publisher>
  <SecurityLevel>1</SecurityLevel>
  <Modules>
    <Module Type="Encrypt">
      <Name>AES</Name>
      <Key type="Symetric">
    126839782069745138402237724462935865545
      </Key>
    </Module>
  </Modules>
</GridStatPolicy>
```

Figure 17 - Publication Policy with key

Similarly, Figure 18 shows the subscription policy returned to the subscriber by the leaf-SMS, when the latter receives a subscription registration request from the former. In this, as in the previous case, the same placeholder $0 is added by the leaf-SMS instead of the key. The Policy Engine at the Subscriber retrieves the key from the key-service by sending a read request to the leaf-SMS and inserts it into the subscription policy, shown in Figure 19. This new policy is exactly the same policy that would have been returned by the leaf-SMS in the absence of the *ProFokus* service integration.

```
<GridStatPolicy>
  <Type>Subscription</Type>
  <Modules>
    <Module Type="Encrypt">
      <Name>AES</Name>
      <Key>$0</Key>
    </Module>
  </Modules>
  <PolicyNum>1</PolicyNum>
  <Publication>test_int_10</Publication>
  <PublicationId>-817033956</PublicationId>
  <Subscriber>gridsim.wsu-sub1</Subscriber>
  <Publisher>gridsim.stn0</Publisher>
  <SecurityLevel>1</SecurityLevel>
</GridStatPolicy>
```

Figure 18 - Subscription Policy with placeholder

```
<GridStatPolicy>
  <Type>Subscription</Type>
  <Modules>
    <Module Type="Encrypt">
      <Name>AES</Name>
      <Key>
    126839782069745138402237724462935865545
      </Key>
    </Module>
  </Modules>
  <PolicyNum>1</PolicyNum>
  <Publication>test_int_10</Publication>
  <PublicationId>-817033956</PublicationId>
  <Subscriber>gridsim.wsu-sub1</Subscriber>
  <Publisher>gridsim.stn0</Publisher>
  <SecurityLevel>1</SecurityLevel>
</GridStatPolicy>
```

Figure 19 - Subscription Policy with key

5.3 Conclusion

The integration of the *ProFokus* key-service with GridStat helped to address an important concern with the framework, that being the confidential fault-tolerant storage of publication keys. Some components like the Publisher and the Subscriber needed to be extended with new functionality, while some functionality was stripped out of components like the leaf-SMS.

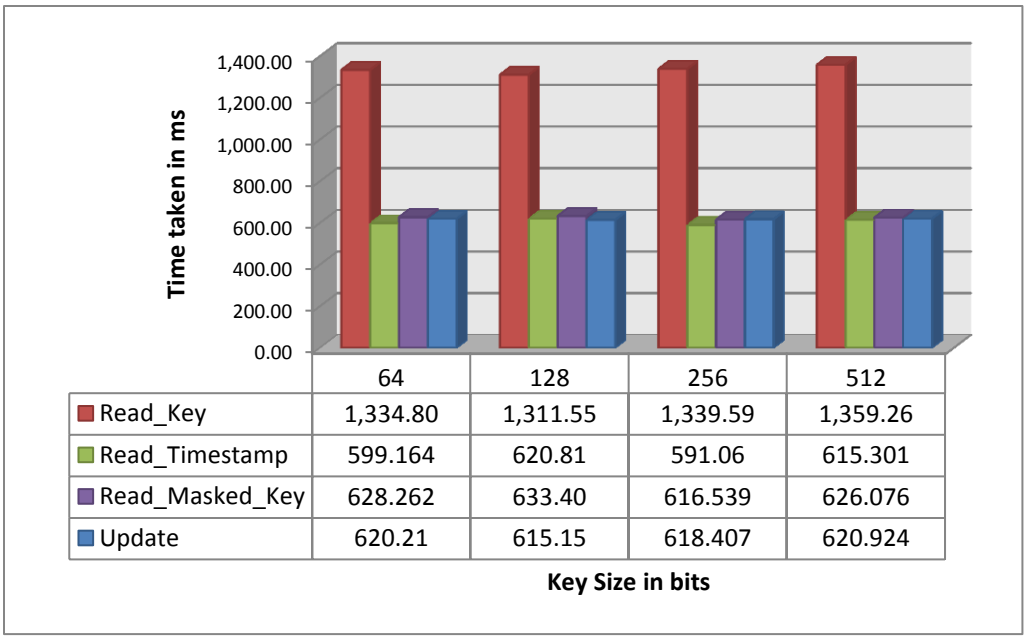
CHAPTER SIX - PROFOCUS IMPLEMENTATION AND PERFORMANCE MEASUREMENTS

The $ProFokus$ system has been implemented in approximately 15K lines of JAVA source. This includes all the protocols, threshold cryptography and proactive recovery. For threshold cryptography, we adapt the library found at $http://threshsig.sf/$, which is an implementation of a threshold RSA scheme [21] for threshold signatures and threshold decryption and this scheme has been configured to use 1024-bits RSA keys. For the server public-private key pairs, 1024-bit RSA keys are used to sign the messages that are sent by the servers. For proactive recovery, we have completely implemented the rejuvenation procedure F that does the work of state recovery, threshold share refresh and payload restart. The implementations of the "Periodic Timely Rejuvenation Service" to periodically invoke the rejuvenation procedures and the physical/logical separation of the PRW from the payload servers are left as part of future work. However their absence has no effect on the $ProFokus$ system or the rejuvenation procedure.

The experiments are designed to observe the latencies in the execution of the $ProFokus$ protocols in the presence and absence of failures and to observe the cost of performing the various cryptographic operations in the protocols. The PlanetLab [31] network was used to run the experiments. Each $ProFokus$ server was deployed on a system with 2-4 cores running Fedora 8 and located at University of Central Florida (UCF). Average round trip times for ICMP echo packets between the servers at the beginning of the experiments were below 1 ms, so the protocol execution times were not

affected by network latencies. The client was executed on a separate host (also located at UCF) but its processing times are not included in the measurements.

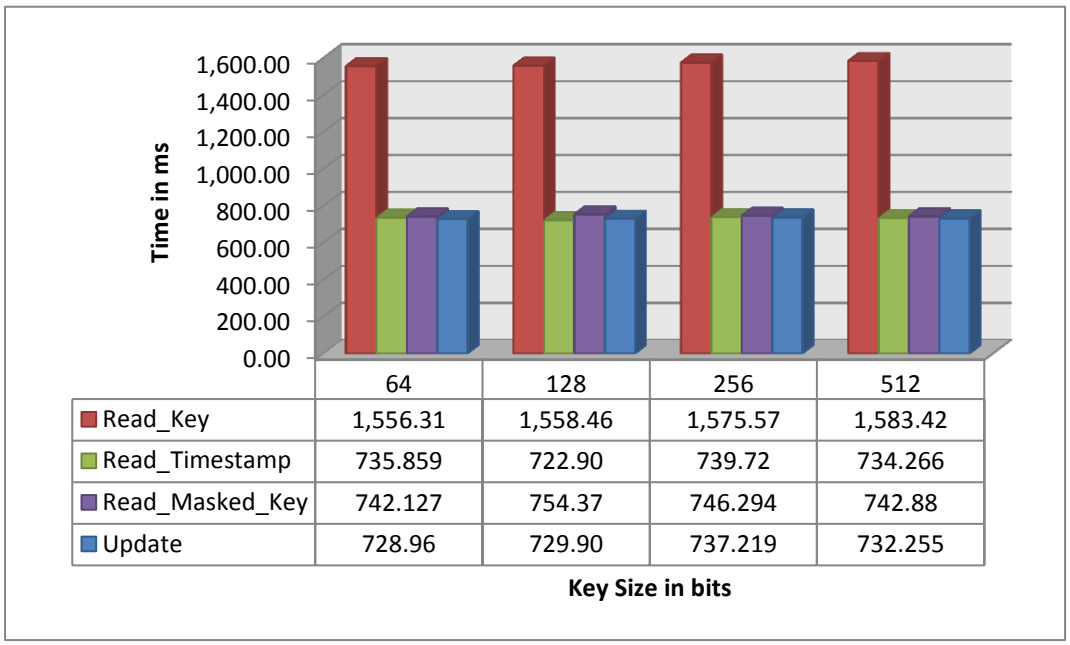
The performance of the *ProFokus* operations is evaluated with fault-tolerance of 1 and 2 respectively. The statistics for each of the measurements come from 100 requests



	64	128	256	512
Read_Key	1,334.80	1,311.55	1,339.59	1,359.26
Read_Timestamp	599.164	620.81	591.06	615.301
Read_Masked_Key	628.262	633.40	616.539	626.076
Update	620.21	615.15	618.407	620.924

Figure 20 - Performance of *ProFokus* Operations with 5 servers, fault-threshold=1

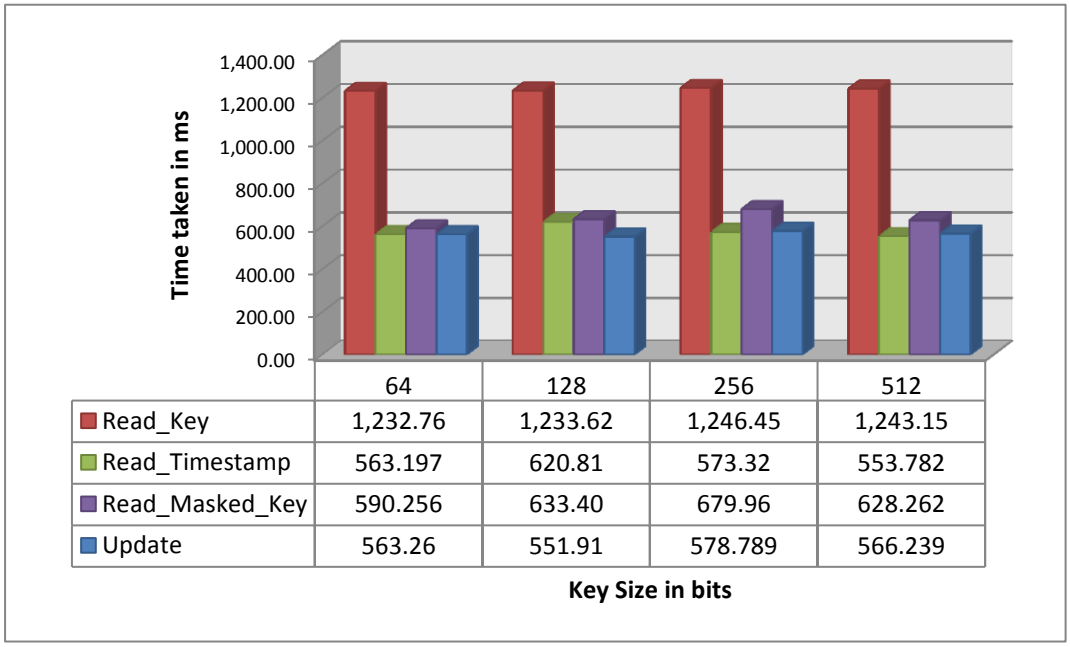
and all times are measured at the delegate. The measurements are repeated for different key lengths ranging from 64 bits to 512 bits. In case of *read_key* operations, the blinding factor length is kept equal to the key length in each of the experiments. Figure 20 gives the mean execution times of the operations for fault-threshold of 1, while gives the times for fault-threshold of 2.



	64	128	256	512
Read_Key	1,556.31	1,558.46	1,575.57	1,583.42
Read_Timestamp	735.859	722.90	739.72	734.266
Read_Masked_Key	742.127	754.37	746.294	742.88
Update	728.96	729.90	737.219	732.255

Key Size in bits

Figure 21 - Performance of *ProFokus* Operations with 10 servers, fault threshold = 2



	64	128	256	512
Read_Key	1,232.76	1,233.62	1,246.45	1,243.15
Read_Timestamp	563.197	620.81	573.32	553.782
Read_Masked_Key	590.256	633.40	679.96	628.262
Update	563.26	551.91	578.789	566.239

Key Size in bits

Figure 22 - Performance of *ProFokus* Operations with 5 servers, 1 faulty server

An adversary can launch a number of attacks on the key-storage service. We consider the impact of the crash-failure of some servers on the performance of the operations. Figure 22 shows the measurements for the operations for a system with 5 servers, in which a single server has suffered a crash failure. Since the *ProFokus* system with 5 servers is designed to tolerate a single fault, there are no visible effects on the latencies of the operations.

	Update	Read_Key	Read_Timestamp	Read_Masked_Key
Message Signing	30.71	29.28	32.17	31.96
Partial Sign Generation	210.46	212.71	210.51	218.38
Partial Sign Verification	332.27	321.06	275.66	347.629
Threshold Sign Generation	2.80	2.66	2.74	2.89
Partial Decryption	0.00	215.64	0.00	0.00
Partial Decryption Verificaton	0.00	506.2	0.00	0.00
Threshold Decryption	0.00	2.59252	0.00	0.00
Other	45.93	53.66	109.60	51.846
Total	622.17	1343.8	630.67	652.705

Table 2 – Time taken by Individual steps in each *ProFokus* Operation in ms

All of the *ProFokus* protocols involve a number of cryptographic operations to implement their functionality:

- **Message Signing**: This includes the cost of digitally signing the messages before sending them to other servers and the cost of verifying the digital signature when a server receives a message.
- **Partial Signature Generation**: In this step, a server generates a partial signature share of a response message using its key-share, when some delegate sends a sign request.
- **Partial Signature Verification**: In this step, the delegate verifies the partial signature shares received from other servers.
- **Threshold Sign Generation**: In this step, the delegate generates a combined signature on the response, using $t + 1$ correct signature shares received from other servers.

- **Partial Decryption**: This step is the same as the **Partial Signature Generation**, except instead of a partial signature, a partial decryption of the stored key is done and sent back to the delegate. This step (and the next two steps) is only performed in the $read\_key$ protocol.
- **Partial Decryption Verification**: In this step, the delegate verifies the partial decryption shares received from other servers.
- **Threshold Decryption**: In this step, the delegate generates the decrypted key-value, using $t + 1$ correct decryption shares received from other servers.

The costs of these cryptographic operations were measured individually and are reported in Table 2. It shows the measurements for a 256 bits key-length and 100 client requests. The times are measured at each delegate and then aggregated. The data is also represented graphically in Figure 23. The results show that the delegate spends most of the time working with threshold signatures in the $read\_timestamp,\ read\_masked\_key\ and\ update$ protocols. There is no threshold decryption required in these protocols and hence these protocols run much faster than the $read\_key$ protocol. In the case of the latter, the threshold decryption operation takes almost as much time as the threshold signature and hence the protocol runs in almost twice the time as compared to the other protocols.

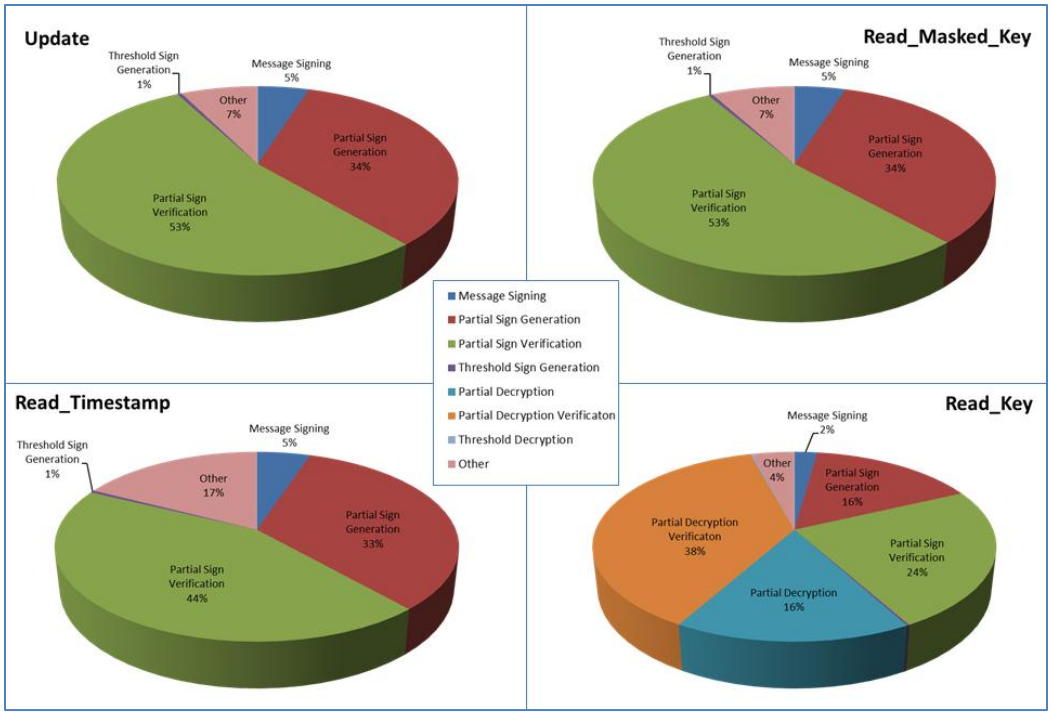


Figure 23 - Cost of Individual steps in each *ProFokus* Operation

CHAPTER SEVEN - CONCLUSION AND FUTURE WORK

This thesis describes the design and implementation of a key-storage service with the ability that can handle an unlimited number of failures over the entire lifetime of the system, provided no more than a certain number of failures occur in a given window. The service framework achieves this by incorporating the following components. Future work required for each of these components is also mentioned.

Byzantine Quorum System: The system is designed using the Byzantine quorum model [14], specifically the $opaque\ masking$ quorum model, which allows the system to maintain a consistent and correct global state despite the failure of some of the server replicas. Such a system model also helps to improve the performance of the system, since only a subset of the servers are involved in handling each request. The intersection between the subsets ensures that the updated state will be returned for a given read request.

Threshold Cryptography: The quorum system model helps to achieve availability and performance, but does not ensure that the stored information is kept confidential. In fact, since the information is held by multiple servers, it works against confidentiality. Threshold cryptography is employed in the $ProFokus$ system and using this scheme, the secret keys are stored in an encrypted form, with their decryption requiring collusion between more servers than the tolerated number of failures. Also, the responses sent to the client are signed using threshold digital signatures and this allows the client to verify them. The experimental results show that the threshold cryptographic operations are

costly, but the use of the Byzantine quorum system ensures that these operations are performed only the minimum required number of times.

Proactive Recovery: With the help of proactive recovery, the system can be made to handle an unlimited number of failures and this is especially important for systems with a long lifetime. The blueprint for the possibility of incorporating Proactive Recovery with the $ProFokus$ system has been presented in this thesis. However, work needs to done to isolate the Proactive Recovery Subsystem from the payload system, either physically or logically. The rejuvenation procedure is dependent partly on the threshold cryptography protocols used and the procedure for RSA threshold cryptography [20] has been implemented in code. The implementations of the "Periodic Timely Rejuvenation Service" to periodically invoke the rejuvenation procedures and the physical/logical separation of the PRW from the payload servers are still to be implemented. Also, a complete evaluation of this recovery procedure needs to be performed.

Proof of Plaintext Knowledge: The $update$ and the $read\_key$ invocations must include proofs of plaintext knowledge [11] so as to prevent an adversary from learning the keys stored in the system. Currently, this is not implemented in code and work needs to be done to incorporate the same.

GridStat: The thesis also presents an application of the $ProFokus$ system by integrating it with a publish-subscribe middleware called GridStat. Publishers store the cryptographic keys that they use to secure the published status information, in the

$ProFokus$ system, while the subscribers retrieve the same keys from the system and use it decrypt or verify the received status information. With masking, the $ProFokus$ system facilitates the storage of keys of any length, thus making the integration with GridStat seamless.

APPENDIX

This chapter gives details about the protocols described in Section 3.4. The client sends a request for read/update to a single server p at first. A server p starts acting as a delegate when it receives the client request or when it receives any message related to the processing of the request.

The following notational conventions are used in the protocols:

$\boldsymbol{p, q}$ **:** $ProFokus$ server

$\boldsymbol{c}$**:** $ProFokus$ client

$< m >_{\boldsymbol{ProFocus}}$**:** message m signed by the $ProFokus$ system using its service private key

$< m >_{\boldsymbol{p}}$**:** message m signed by a server p using its own private key,

$< m >_{\boldsymbol{c}}$**:** message m signed by a client c using its own private key,

$\boldsymbol{\phi(m, s_p)}$**):** a partial signature for a message m generated by a server p using its own partial share s_p of the service private key,

$\boldsymbol{p \rightarrow q: m}$**:** message m is sent from host p to host q

$\forall \boldsymbol{q}, \boldsymbol{p \rightarrow q: m_q}$: message m_q is sent from server p to server q for each server q.

Each message includes a type indicator to indicate the purpose of the message. These type indicators are presented in the sans serif font.

$Read\_Key$ Protocol

1. To read the key value for name N, A client $\boldsymbol{c}$ sends invocation message to $t+1$ delegates

$$Req_{key} : < read, N, E(b_c), \pi(b_c, c) >_c$$

where $\boldsymbol{E(b_c)}$: the blinding factor encrypted using $ProFokus$ public key

$\pi(b_c, c)$: proof of plaintext knowledge of b_c

2. The client waits for valid $key\_response$ messages from any of the $ProFokus$ servers. If the client times out, then it sends the same request to $t+1$ delegates and waits again.

Each delegate D upon receiving Req_{key} proceeds as follows:

1. D checks the validity of Req_{key} by checking its signature, checking the validity of $E(b_c)$ and checking the validity of proof of plaintext knowledge $\pi(b_c, c)$
2. If Req_{key} is invalid, then D ignores the request. If valid, then D forwards Req_{key} along with a nonce n to all the $ProFokus$ servers S_i

$$\forall i, D \rightarrow S_i : n, Req_{key}$$

using a repeated send primitive and awaits until it gets responses from a quorum of servers with the same timestamp.

3. Upon receipt of a message n, Req_{key} from D, server S_i applies checks of step 1.
    a. If Req_{key} is invalid, then S_i ignores the request. Also, it adds D to its list of compromised servers since it sent an invalid client request.
    b. If Req_{key} is valid, then S_i retrieves the binding $\beta =< \tau_N, N, E(k_N) >$ for name N. If no such binding exists, then S_i ignores the request. S_i then

calculates the blinded ciphertext $c_i = E(k_N * b_c) = E(k_N) * E(b_c)$, its partial decryption $D_i(c_i)$ and proof DL_i of the validity of $D_i(c_i)$, using threshold decryption and its share of the private service key. This is sent back along with the timestamp τ_N that it currently stores for name N:

$$S_i \rightarrow D :< n, key\_response, \tau_N, c_i, D_i(c_i), DL_i , Req_{key} >_{S_i}$$

4. A $key\_response$ message is deemed valid by D if it is correctly signed and DL_i is valid.
    a. When a quorum number of valid messages have been received, then D chooses a set ξ such that each message in that set has the same timestamp and the same ciphertext. In case some servers are compromised, then two such sets might exist with the same number of servers. In such a case, D chooses the set with the higher timestamp.
    b. D uses $(t + 1)$ shares from the set ξ to obtain the decrypted value $k_N * b_c$. D creates the response $\Re^-$ that it will send back to the client as follows :

$$\Re^-: key\_response , N, k_N * b_c, Req_{key}$$

    c. D invokes a threshold signature protocol with all servers to sign $\Re^-$ using the set ξ as evidence
        i. $\forall i, D \rightarrow S_i : \xi_D, \Re^-$
        ii. D awaits valid partial signatures $\phi(\Re^-, S_i)$ from $(t + 1)$ servers and uses those signatures to obtain the signed response $\Re_{key}$

$$\Re_{key} :< \Re^- >_{ProFokus}$$

        iii. $D \rightarrow c : \Re_{key}$

$Read\_timestamp$ Protocol

To do an update operation, A client c needs to first get its current timestamp from $ProFokus$

1. A client $\boldsymbol{c}$ sends invocation message to $t + 1$ delegates

$$Req_{TS} : < read\_timestamp, N >_c$$

2. The client waits for valid $timestamp\_response$ messages from any of the $ProFokus$ servers. If the client times out, then it sends the same request Req_{TS} to $t + 1$ delegates and waits again.

Each delegate D upon receiving Req_{TS} proceeds as follows:

1. D checks the validity of Req_{TS} by checking the signature on Req_{TS}
2. If Req_{TS} is invalid, then D ignores the request. If valid, then D forwards Req_{TS} along with a nonce n to all the $ProFokus$ servers S_i

$$\forall i, D \rightarrow S_i : n, Req_{TS}$$

using a repeated send primitive and awaits responses from a quorum of servers.

3. Upon receipt of a message n, Req_{TS} from D, a server S_i applies checks of step 1.
    a. If Req_{TS} is invalid, then S_i ignores the request. Also, it adds D to its list of compromised servers since it sent an invalid client request.
    b. If Req_{TS} is valid, then S_i returns the current timestamp τ_N that it currently stores for name N. If no such name exists, then it returns a value of 1.

$$S_i \rightarrow D :< n, timestamp\_response, \tau_N, Req_{TS} >_{S_i}$$

4. A $timestamp\_response$ message is deemed valid by D if it is correctly signed.
    a. When a quorum number of valid messages have been received, then D chooses a set ξ such that each message in that set has the same timestamp.

In case some servers are compromised, then two such sets might exist with the same number of servers. In such a case, D chooses the set with the higher timestamp.

b. D creates the response $\Re^-$ that it will send back to the client as follows :

$$\Re^- : timestamp\_response, N, \tau_N, Req_{TS}$$

c. D invokes a threshold signature protocol with all server to sign $\Re^-$ using the set ξ as evidence.

i. $\forall i, D \rightarrow S_i : \xi_D, \Re^-$

ii. D awaits valid partial signatures $\phi(\Re^-, S_i)$ from $(t+1)$ servers and uses those signatures to obtain the signed response $\Re_{TS}$

$$\Re_{TS} : < \Re^- >_{ProFokus}$$

iii. $D \rightarrow c : \Re_{TS}$

Update **Protocol**

1. To update the key value associated with a name $\boldsymbol{N}$, A client $\boldsymbol{c}$ gets the signed timestamp response from $\boldsymbol{ProFokus}$ using the $\boldsymbol{read\_timestamp}$ protocol:

$$\Re_{TS} : < timestamp\_response, N, \tau, Req_{TS} >_{ProFokus}$$

2. The client sends invocation message to $\boldsymbol{t+1}$ delegates

$$Req_{update} : < update, N, E(k), \Re_{TS}, \pi(k, c) >_c$$

where $\boldsymbol{E(k)}$: the secret key encrypted using $ProFokus$ public key

$\pi(k, c)$: proof of plaintext knowledge of k

3. The client waits for a valid $updated$ confirmation response from any of the $ProFokus$ servers. If the client times out, then the client restarts from step 1.

Each delegate D upon receiving Req_{update} proceeds as follows :

1. D checks the validity of Req_{update} by checking its signature, checking the signature on $\Re_{TS}$ and checking the proof of plaintext knowledge $\pi(k, c)$
2. If Req_{update} is invalid, then D ignores the request. If valid, then D forwards Req_{update} along with a nonce n to all the $ProFokus$ servers S_i

$$\forall i, D \rightarrow S_i : n, Req_{update}$$

using a repeated send primitive and awaits acknowledgements from a quorum of servers.

3. Upon receipt of message n, Req_{update} from D, server S_i applies checks of step 1.
    a. If Req_{update} is invalid, then S_i ignores the request. Also, it adds D to its list of compromised servers since it sent an invalid client request.
    b. If Req_{update} is valid, then S_i extracts the timestamp τ from $\Re_{TS}$ and checks if $\tau \geq \tau_{local(N)}$
        i. If yes, then S_i updates the timestamp and the key-value for N with the values from Req_{update} and also stores the hash of the request message, $hash_N = hash(Req_{update})$. It also replies

$$S_i \rightarrow D :< n, update\_accept, Req_{update} >_{S_i}$$

        ii. If no, then S_i checks if the stored $hash_N = hash\left(Req_{update}\right)$

a) If yes, then S_i replies with the same message as in step i.

b) Else S_i ignores the request.

4. An $update\_accept$ message is deemed valid by D if it is correctly signed. Each such valid message is added to an set ξ.

   a. When a quorum pieces of valid $update\_accept$ messages have been collected, D creates the response $\Re^-$ that it will send back to the client as follows :

$$\Re^- : updated, N, Req_{update}$$

   b. D invokes a threshold signature protocol with all server to sign $\Re^-$ using the set ξ as evidence

      i. $\forall i, D \rightarrow S_i : \xi_D, \Re^-$

      ii. D awaits valid partial signatures $\phi(\Re^-, S_i)$ from $(t+1)$ servers and uses those signatures to obtain the signed response $\Re_{update}$

$$\Re_{update} : < \Re^- >_{ProFokus}$$

      iii. D forwards the response to all the other servers

$$\forall i, D \rightarrow S_i : n, \Re_{update}, Req_{update}$$

      iv. $D \rightarrow c : \Re_{update}$

5. Upon receipt of message $n, \Re_{update}, Req_{update}$ from D, server S_i checks:

   a. if the threshold signature on $\Re_{update}$ is valid,

   b. applies the checks of step 1 for Req_{update}

If all the above hold and if server S_i is currently a delegate or is planning to become a delegate for Req_{update}, then it stops acting as one. Also, S_i sends the response $\Re_{update}$ back to the client.

$\boldsymbol{Read\_Masked\_Key}$ Protocol

This protocol is used to get the masked key value for a name N.

1. A client $\boldsymbol{c}$ sends invocation message to $t + 1$ delegates

$$Req_{MK} : < read\_masked\_key, N >_c$$

2. The client waits for valid a $masked\_key\_response$ message from any of the $ProFokus$ servers. If the client times out, then it sends the same request Req_{MK} to $t + 1$ delegates and waits again.

Each delegate D upon receiving Req_{MK} proceeds as follows:

1. D checks the validity of Req_{MK} by checking the signature on Req_{MK}
2. If Req_{MK} is invalid, then D ignores the request. If valid, then D forwards Req_{MK} along with a nonce n to all the $ProFokus$ servers S_i

$$\forall i, D \rightarrow S_i : n, Req_{MK}$$

   using a repeated send primitive and awaits responses from a quorum of servers.

3. Upon receipt of a message n, Req_{MK} from D, a server S_i applies checks of step 1.
   a. If Req_{TS} is invalid, then S_i ignores the request. Also, it adds D to its list of compromised servers since it sent an invalid client request.
   b. If Req_{MK} is valid, then S_i retrieves the binding $\beta = < \tau_N, N, E(k_N) >$ it stores for name N and returns that. If no such name exists, then it ignores the request.

$$S_i \rightarrow D :< n, masked\_key\_response, E(k_N), \tau_N, Req_{MK} >_{S_i}$$

4. A $masked\_key\_response$ message is deemed valid by D if it is correctly signed.

a. When a quorum number of valid messages have been received, then D chooses a set ξ such that each message in that set has the same timestamp and the same key-value $E(k_N)$. In case some servers are compromised, then two such sets might exist with the same number of servers. In such a case, D chooses the set with the higher timestamp.

b. D creates the response $\Re^-$ that it will send back to the client as follows :

$$\Re^- : masked\_key\_response, N, E(k_N), Req_{MK}$$

c. D invokes a threshold signature protocol with all server to sign $\Re^-$.

iv. $\forall i, D \rightarrow S_i : \xi_D, \Re^-$

v. D awaits valid partial signatures $\phi(\Re^-, S_i)$ from $(t+1)$ servers and uses those signatures to obtain the signed response $\Re_{MK}$

$$\Re_{MK} : < \Re^- >_{ProFokus}$$

vi. $D \rightarrow c : \Re_{MK}$

REFERENCES

[1] Bakken, D., Hauser, C., and Gjermundrød, H. "Periodically Updated Variables: Wide-Area Publish-Subscribe Middleware Supporting Electric Power Monitoring and Control". *Paper submitted for publication to the 2009 IEEE International Conference on Distributed Computing Systems (ICDCS2009)*. Available: http://www.eecs.wsu.edu/~bakken/ICDCS09Submission-GridStatOverall.pdf

[2] Bakken, D., Hauser, C., Gjermundrød, H. and Bose, A. "Towards More Flexible and Robust Data Delivery for Monitoring and Control of the Electric Power Grid", Technical Report EECS-GS-009, Washington State University, May 2007. Available: http://www.gridstat.net/trac/raw-attachment/wiki/publications/TR-GS-009.pdf

[3] Barham, P., Dragovic, B., Fraser, K., et al. "Xen and the art of virtualization". *Proceedings of the nineteenth ACM symposium on Operating systems principles, NY, USA, ACM* 2003, 164–177.

[4] Chakravarthy, R., Hauser, C., and Bakken, D.E. "Long-lived authentication protocols for process control systems". *International Journal of Critical Infrastructure Protection 3*, 3-4 (2010), 174–181.

[5] Kelly, J., McVittie, T. and Yamamoto, W. "Implementing design diversity to achieve fault tolerance". *IEEE Software*, vol. 8, Jul. 1991, pp. 61 –71

[6] Clements S. L. 2009. "GridStat – Cyber Security and Regional Deployment Project Report". PNNL-18252, Pacific Northwest National Laboratory, Richland, WA.

[7] Desmedt, Y. "Some Recent Research Aspects of Threshold Cryptography". *Proceedings of the First International Workshop on Information Security*, Springer-Verlag (1998), 158–173.

[8] Desmedt, Y.G. "Threshold cryptography". *European Transactions on Telecommunications 5*, 4 (1994), 449–458.

[9] Fischer, M. and Lynch, N. "Impossibility of distributed consensus with one faulty process". *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[10] Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M. "Proactive Secret Sharing Or: How to Cope With Perpetual Leakage". *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, Springer-Verlag (1995), 339–352.

[11] Katz, J. "Efficient and non-malleable proofs of plaintext knowledge and applications". *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, Springer-Verlag (2003), 211–228.

[12] Lamport, L., Shostak, R., and Pease, M. "The Byzantine Generals Problem". *ACM Transactions on Programming Languages and Systems (TOPLAS). 4*, 3 (1982), 382–401.

[13] Malkhi, D. and Reiter, M. "Byzantine Quorum Systems". *Distributed Computing 11*, 4, 203–213.

[14] Marsh, M. and Schneider, F. "CODEX: A robust and secure secret distribution system". *IEEE Transactions on Dependable and Secure Computing, 1*, 1 (2004), 34–47.

[15] Ostrovsky, R. and Yung, M. "How to withstand mobile virus attacks (extended abstract)". *Proceedings of the tenth annual ACM symposium on Principles of Distributed Computing, ACM* (1991), 51–59.

[16] Rivest, R.L., Shamir, A., and Adleman, L. "A method for obtaining digital signatures and public-key cryptosystems". *Communications of the ACM* 21, 2 (1978), 120–126.

[17] Rodrigues, R. and Liskov, B. "Byzantine Fault Tolerance in Long-Lived Systems." *2nd Bertinoro Workshop on Future Directions in Distributed Computing (FuDiCo II)*, (2004) Bertinoro, Italy.

[18] Schneier, B. "Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition." *Wiley*, 1996.

[19] Shamir, A. "How to share a secret". *Communications of the ACM 22*, 11 (1979), 612–613.

[20] Shoup, V. "Practical threshold signatures". *Proceedings of the 19th international conference on Theory and application of cryptographic techniques*, Springer-Verlag (2000), 207–220.

[21] Solum E., Hauser C., and Bakken D. E., "Modular Over-The-Wire Security in Managed Publish-Subscribe Systems". Technical Report EECS-GS-010, School of Electrical Engineering and Computer Science, Washington State University, December 2007. Available: http://www.gridstat.net/trac/raw-attachment/wiki/publications/TR-GS-010.pdf

[22] Sousa, P., Neves, N.F., and Verissimo, P. "Hidden problems of asynchronous proactive recovery". *Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, USENIX Association (2007).

[23] Sousa, P., Neves, N.F., and Veríssimo, P. "A New Approach to Proactive Recovery". *Fifth European Dependable Computing Conference (EDCC-5) Supplementary Volume*, (2005), 35–40.

[24] Sousa, P., Neves, N.F., and Veríssimo, P. "Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available". *In Proc. of the Symp. on Reliable Distributed Systems*, IEEE Computer Society (2006), 71–80.

[25] Tilborg, H.C.A. van. "Encyclopedia of Cryptography and Security". Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[26] Wylie, J.J., Bigrigg, M.W., Strunk, J.D., Ganger, G.R., Kiliççöte, H., and Khosla, P.K. "Survivable Information Storage Systems". *Computer 33*, 8 (2000), 61–68.

[27] Zhou, L., Schneider, F.B., and Van Renesse, R. "COCA: A secure distributed online certification authority". *ACM Transactions on Computer Systems 20*, 4 (2002), 329–368.

[28] Zhou, L., Schneider, F.B., and Van Renesse, R. "APSS: Proactive secret sharing in asynchronous systems". *ACM Transactions on Information and System Security (TISSEC) 8*, 3 (2005), 259–286.

[29] Sousa, P. "How Resilient are Distributed f Fault/Intrusion-Tolerant Systems?" *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, IEEE Computer Society (2005), 98–107.

[30] PlanetLab, An open platform for developing, deploying and accessing planetary-scale services. *http://www.planet-lab.org/*.

[31] Kenchington, Hank. "Securing Tomorrow's Grid - Protecting smart systems against cyber threats", *Fortnightly Magazine*, July 2011, 28-42.