ACHIEVING OVER-THE-WIRE CONFIGURABLE CONFIDENTIALITY, INTEGRITY, AUTHENTICATION AND AVAILABILITY IN GRIDSTAT'S STATUS DISSEMINATION

By

ERIK SOLUM

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

DECEMBER 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of ERIK SOLUM find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

First of all I would like to thank my advisor and committee chair Dr. Carl Hauser and committee member Dr. David E. Bakken for their academic and social support during my time here at Washington State University as an international student. The rest of the GridStat group and the larger TCIP group should also be acknowledged for the good collaboration, especially Adam Lee which with his insightful questions kept my research focused.

I would also like to thank my third committee member Dr. Min Sik Kim for taking the time to be on my committee and my fellow, now graduated, students Erlend Viddal and Stian Abelsen for providing a fruitful working environment and helping me with the preparation of my defense.

Finally I would like to thank the organizations that provided financial support for my education and research. My education would not have been possible without the stipends given to me by The Norwegian State Educational Loan Found that has partially supported my whole higher education from my freshman year back in Norway through my time here at WSU. In addition my research have been partially supported by the the grant CNS 05-24695 (CT-CS: Trustworthy Cyber Infrastructure for the Power Grid(TCIP)) and by the U.S. Department of Energy, Office of Electricity Delivery, via subcontract 49944 with Pacific Northwest National Laboratory.

PUBLICATIONS

Erik Solum, Carl Hauser and David Bakken: Modular Over-The-Wire Security in Managed Publish-Subscribe Systems. Submitted to the International Conference on Dependable Systems and Networks (DSN'08) in December 2007.

ACHIEVING OVER-THE-WIRE CONFIGURABLE CONFIDENTIALITY, INTEGRITY, AUTHENTICATION AND AVAILABILITY IN GRIDSTAT'S STATUS DISSEMINATION

Abstract

by Erik Solum, M.S.
Washington State University
December 2007

Chair: Carl Hauser

As a result of ever increasing demands for electrical power the power grid is continuously being operated closer and closer to its operational limits. This can only be done safely by increasing both the quantity and quality of the monitoring data across utilities. GridStat is a framework that tries to address this need by leveraging a QoS aware status dissemination overlay network built on the publish-subscribe paradigm.

The publish-subscribe paradigm allows a decoupling of the producers and the consumers of information. In GridStat the publishers produce status updates at regular intervals, which the subscribers can subscribe to at any rate they need with quality of service (QoS) guarantees, such as maximum latency and redundant paths, at any point in the network. The status updates are routed through a mesh of application level routers called the data plane, controlled by a management plane of hierarchically structured QoS brokers.

The power grid's increasing reliance on richer monitoring data also necessitates a greater level of security, especially considering the world's building political tensions. Sensitive data also needs to be secured from malicious attackers that could use the information indirectly or, by manipulating the data, directly harm the power grid. The inter-utility-sharing of information also makes it necessary to keep market sensitive data confidential from competitors. The real challenge

in this problem space lies in providing the security for power grid information systems that are large and distributed with long life cycles. Unmanned nodes would be expected to operate for as much as 25 years while the security requirements are constantly changing and unpredictable.

This thesis presents a security architecture extension to GridStat's management plane that provides confidentiality, integrity, authentication and availability to the data plane through the use of over-the-wire runtime configurable sets of software modules. New modules can be added to the security architecture at runtime and be securely distributed to the data plane end points. This allow the security to evolve with the inevitable changes in the security field and make optimal tradeoffs between different security and performance attributes for each individual publication.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

To my parents,

for their continuous support.

CHAPTER ONE

INTRODUCTION

The power grid is one of the largest man-made structures in the world. It is a complex structure that spans a multitude of domains and organizations that each can affect it to a greater or lesser extent. In an effort to avoid blackouts such as the 1965 North-eastern US blackouts the *Supervisory Control And Data Acquisition* (SCADA) system was developed and is still the industry standard. This is a static poll based system that does not take advantage of the developments in information technology since the 1960's.

At the same time power consumption has increased tremendously since the 1960's without a corresponding use of resources to increase the the grid's capacity. This increase forces the grid to operate with decreasing margins [22]. As the 2003 US and Canadian blackout exemplified, a new information system that enables greater quantity and quality of sensors, real time monitoring and a higher degree of inter-utility information sharing is needed to handle this new reality [12].

GridStat is an information system being developed to solve many of the problems currently evident in the power grid that up to now mainly has implemented performance and fault-tolerance considerations. This thesis presents a security architecture that provides GridStat's data dissemination with confidentiality and integrity in addition to further enhance its availability.

1.1 GridStat

GridStat is a middleware approach to a data acquisition framework under development for the power grid that uses a unique combination of distributed network technologies to provide fault-tolerance combined with real time performance [21]. It employs a specialized version of the *publish-subscribe* paradigm centered on *status dissemination*. It is managed by a *management plane*, that provides it with *quality of service* (QoS) guarantees. Status dissemination is based upon periodic updates of *status variables* called *status updates* that are routed to the subscribers at

constant, but differentiated rates.

In GridStat the routing of status updates are done by application-level routers, called *status routers*, comprising an overlay network topology on top of any legacy network infrastructure. This enables GridStat to span a heterogeneous set of underlying network protocols and, with the help of the management plane, extend end-to-end QoS guarantees across them in a way no native network protocol could [18]. The publishers, subscribers and status routers are defined as the *data plane* to differentiate it from the management plane which controls it.

The latest extensions to GridStat is the introduction of hierarchical modes that enables quick switches from one subset of subscriptions to another to handle power grid contingencies [10]. Mode switches can also be used in cases of denial of service attacks to shead data. The mode switch extension uses a remote procedure call (RPC) framework called *Ratatoskr*, another new addition developed on top of GridStat's data dissemination[35]. By building an RPC mechanisms on top of the publish-subscribe data dissemination Ratatoskr inherits, and the hierarchical modes by transitivity, its properties such as QoS guarantees. The security architecture presented in this thesis will provide these extensions with security through securing the data plane that they rely on.

1.2 Securing GridStat's Data Plane

Any system aiming to transport sensitive data need to make efforts to secure that data. GridStat being developed for disseminating status information in the power grid makes it not only imperative to ensure confidentiality, but since safety critical decisions will be based upon this information, also integrity, authentication and availability.

One of GridStat's main goals is to enable a national deployment that spans utilities and thus enable information sharing that can increase the power grid stability [12]. In order for this to be possible the utilities need to feel that their market sensitive data is safe from other utilities. The utilities therefore need to be able to differentiate the confidentiality levels on status information.

In addition to securing market sensitive data from competitors, information systems for the

power grid need to defend against malicious attacks that intend to harm the power grid. The more comprehensive an information system becomes, the greater the consequences of a successful attack and thus the need for security measures increases. In light of the last decade's developments in the world and the "war on terror" the need for securing the power grid against such attacks have been all around recognized [15] and has manifested itself, among other things, as Department of Energy (DOE) and National Science Foundation (NSF) grants for research into this area. GridStat is part of such a research project called Trustworthy Cyber-infrastructure for Power Systems (TCIP) [1].

1.2.1 Problem scope

GridStat has up to now focused on availability, performance and integrity with an emphasis on fault-tolerance aspects. The goal of this thesis is to develop a *Data Plane Security Architecture* that addresses the need for a self-sufficient security system that provides mechanisms which, when combined with later developed policies, ensure confidentiality, integrity, authentication and further improve the accessability of the communication streams between the publishers and subscribers in the data plane. The security architecture also have to handle special GridStat's mechanisms, of which *multicast* (2.1.2.2), *redundant paths* (2.1.2.1) and *rate filtering* (2.1.2) are the most notable, while exploiting some of its other properties, such as its relatively static topology and types of services.

Securing the flow of information in the data plane entails securing the status updates against attacks from the moment they leave the publisher until they are received by the subscriber. These attacks could be everything from simple sniffing, to more advanced man-in-the-middle attacks and injection of bad data.

The problem scope is limited to the communication and does not involve end-point security of the GridStat components such as publishers, subscribers, status routers, QoS brokers and their management communication. Although consequences of breaches in their security will be taken into consideration, security mechanisms for the end-points are defined outside the problem scope

and will have to be addressed by future projects.

In order to make a security system self-sufficient it needs to secure its own communication. Nothing is gained by adding security measures for the data plane while introducing new security weaknesses in the management plane. Any security system needs to protect its own management communication by providing confidentiality, integrity and authentication in the same way as it provides it for the payload data.

1.2.2 Dynamic Security Architecture

Information systems for the power grid have life expectancies of 25 years or more and thus causes another serious technical challenge for this problem space. No one knows how much the computational power available to attackers will increase over such a long period of time, not to mention possible breakthroughs in ways to crack specific algorithms. This makes it very hard to design a static security system that with reasonable certainty can be trusted until the communication system someday is replaced. If using such a static approach is at all possible it would necessitate using a worst case approach when choosing algorithms and key lengths to account for the uncertainty in what the future might bring. This would be extremely wasteful on resources and add unacceptable levels of latency.

To avoid using worst case approaches, a dynamic solution where changes can be done to the security in response to inevitable changes in the security field is desirable, allowing GridStat to evolve with the changes in the security field instead of being at its mercy.

1.2.3 Challenges

The fact that information systems for the power grid are large distributed networks of mainly unmanned nodes and that crucial control decisions relays on the data from these nodes add some additional challenges with the use of dynamic approaches. Since no information can be lost it is impossible to shut down nodes for maintenance. Updates have to be done in runtime without any significant interruption in the information flow. The potential remoteness of the nodes make

manual updates too inconvenient to be a viable solution and necessitates a solution where the nodes can be remotely updated.

On top of the more general challenges of life expectancy and node distribution the fact that GridStat employs techniques such as redundant paths and multicast places further restrictions on the development of a security architecture for the data plane. Multicast is based upon the idea that identical packets are being sent to multiple recipients. This makes it impractical to use any type of security measures that differentiate the packets being sent based upon their recipients. The use of differentiated rates between the publishers and subscribers also invalidates the use of stream ciphers since they are based upon a stream where the producer and consumers are synchronized.

Bounding the latency that is introduced to acceptable levels also poses a challenge. Communication systems for the power grid need to provide status updates in real-time. Real-time requirements heavily favor an end-to-end approach where the added latency of a security measure is bounded by the end point latencies, and not a function of the length of the path.

Finally the real world problem of the utilities having concerns against letting others control the access to their information has to be addressed. For GridStat to be successfully deployed the utilities need to feel that their market sensitive data is safe. This not only means encrypting to achieve data confidentiality, but also keeping the access control of a utilities data within the utility or else the utilities would be reluctant to move from the current system where they have complete control.

1.3 Research Contributions

The research contributions of this thesis are:

- Design and implementation of a novel modular security architecture for the data plane of managed status dissemination publish-subscribe frameworks (GridStat).

- Design and implementation of a protocol that allows secure redistribution of security modules and keys over compromised communication links to reestablish security.
- Exploration of the different types of security modules relevant for information systems for the power grid and a series of proof-of-concept module implementations based on those explorations.
- An experimental performance evaluation of the security architecture's infrastructure as well as the proof-of-concept modules, individually and in sets.

1.4 Organization of Thesis

The thesis is divided into six chapters, Chapter 1 has introduced the problem space and goals of the thesis while Chapter 2 delves into more background details such as the GridStat framework, the current standards in the power grid industry and existing research on adding security to the publish-subscribe paradigm.

Chapter 3 presents a threat model based on the industry and academic background research explored in Chapter 2 before it moves on to a novel security architecture addressing the threats through the use of over-the-wire configurable sets of software modules. Then Chapter 4 specifies the design and implementation of the security architecture and a series of proof-of-concept modules.

The performance of the implementation of the security architecture is evaluated in Chapter 5, both with respect to scalability of the infrastructure and the real-time capabilities of the data plane. Finally Chapter 6 offers concluding remarks and potential areas of future research.

CHAPTER TWO

BACKGROUND AND RELATED WORK

This Chapter provides a background to the security architecture presented in this thesis. Section 2.1 gives an overview of GridStat, its properties, restrictions and challenges while Section 2.2 looks into the current industry standards and requirements. Finally, conventional PKI approaches and current publish-subscribe security research are described in Section 2.3 and 2.4 respectively.

2.1 GridStat

GridStat is a rate-based status dissemination network developed mainly for use in the power grid, but it has many other applicable domains where the majority of the data is periodic in nature. It 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. By combining publish-subscribe with QoS management in a novel way GridStat can disseminate periodic status information to a multitude of parties in real time with scalability, flexibility, timeliness and reliability [21, 12].

Being an overlay network, GridStat can run on top of existing and future communication infrastructures overcoming their inherent heterogeneity. This is accomplished by application-level routers called *status routers*. The status routers are the backbone of the *data plane* which also contains *publishers* and *subscribers*. The status routers are controlled by a *management plane* that provides QoS guarantees to the subscribers through a hierarchy of QoS brokers as shown in Figure 2.1.

GridStat employs a unique combination of techniques to improve reliability, timeliness and performance of which the use of redundant paths, multicast and bandwidth control are some of the most significant. The use of redundant paths not only greatly improves the reliability and average latency, but can significantly reduce the cost of employing multicast.

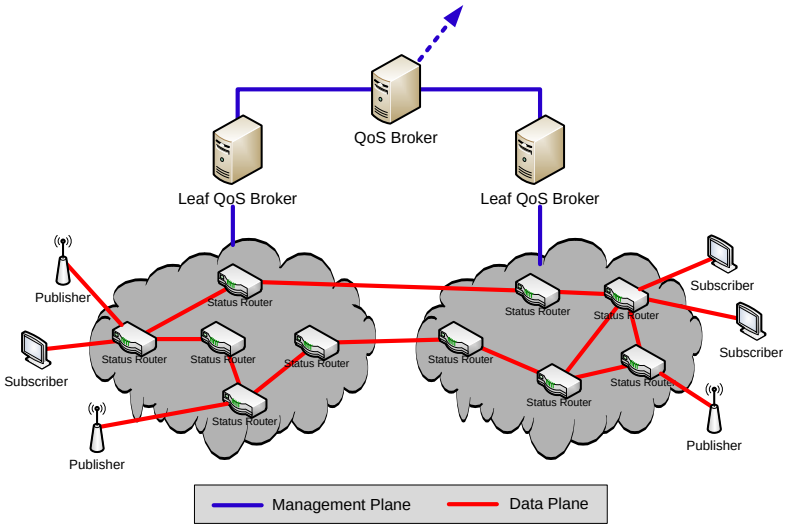

Figure 2.1: GridStat framework overview

2.1.1 Management Plane

The management plane, also called the *QoS broker hierarchy*, is a set of *QoS brokers* organized in a hierarchy that manages the resources of the data plane. Each QoS broker manages registration and deregistration of publications and subscriptions, path allocation, placement of condensation functions, status router control and mode change mechanisms [10] to its subset of the data plane. Figure 2.2 illustrates that there currently are no limit to the number of levels in the hierarchy, and how the lowest level of the hierarchy is populated by leaf QoS brokers, of which subset of the data plane they control are called *clouds*.

To set up a new status variable publications, publishers contact the *leaf QoS broker* that serves their cloud and specify the properties of the variable to be published such as type, size and rate before starting to push updates into the data plane. The leaf QoS brokers propagate this information up through their ancestors to the root QoS broker.

Subscribers that want to subscribe to any of the available publications also contact their leaf QoS broker with the publication name together with the desired QoS attributes for the different modes they want to subscribe to the publication in. The subscription requests are then forwarded

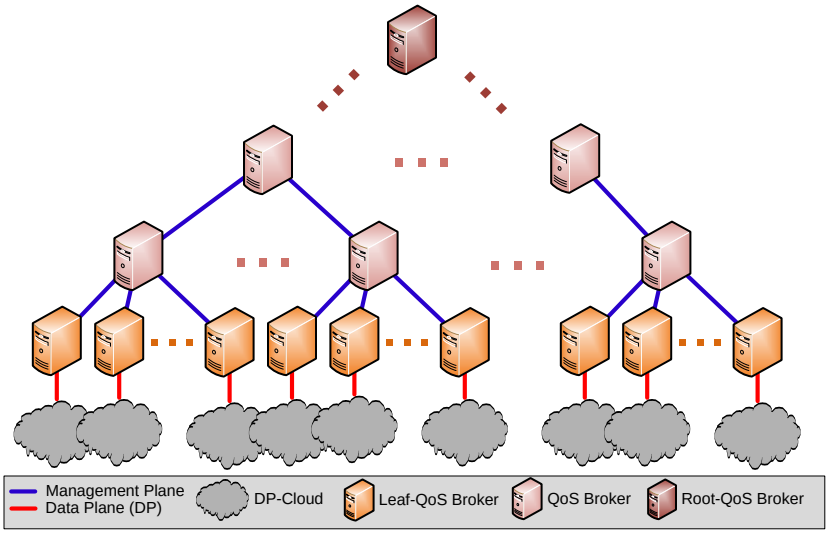

Figure 2.2: The Management Plane

up the hierarchy until they reach the lowest common QoS broker ancestor between the publisher and the subscriber. This QoS Broker decides whether the desired QoS parameters can be met and if so, starts allocating paths between the publisher and subscriber. If the publisher and subscriber share the same leaf QoS broker the paths will be internal to their cloud, otherwise inter-cloud paths will be set up.

The QoS brokers allocate paths in the data plane by directly controlling the routing tables in the status routers. Each status router got a routing table for each of its ancestor QoS brokers controlling separate percentages of the available bandwidth.

2.1.2 Data Plane

The data plane is a virtual message bus that routes status updates from publishers to subscribers based upon application-level status routers organized into clouds, each controlled by a leaf QoS

broker. Figure 2.3 illustrates how the publish-subscribe paradigm enables information to be inserted and extracted at any point in the data plane by adding publishers and subscribers. Subscribers can subscribe to any publication, regardless of the publishers position. In Figure 2.3, the subscribers *Sub-1* and *Sub-2* both subscribe to the same publication published by *Pub-1* as shown by the red lines.

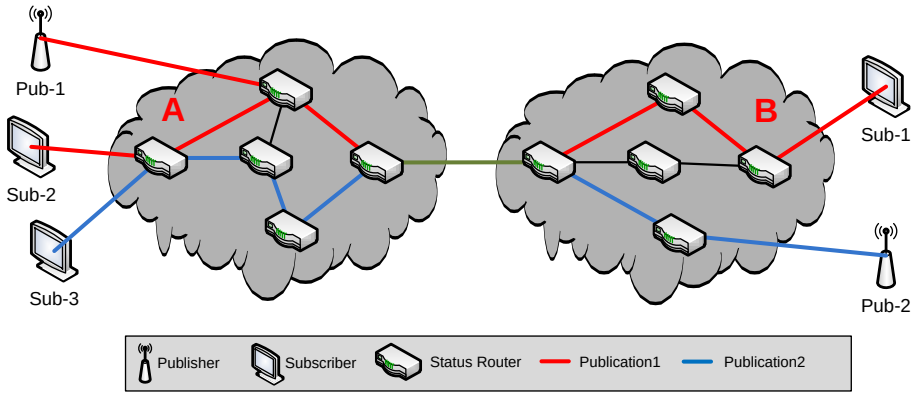

Figure 2.3: The Data Plane

The status routers use rate filtering and multicast to better utilize the bandwidth resources. When subscribers subscribe to a publication, they can freely specify the rate of updates they want as long as it is less or equal to the publication rate. This means that different subscribers with different needs can use different rates on the same publication. The status routers filter all the status updates based upon the current need of subscribers, resulting in drops of all unneeded status updates. In the extreme case of there not being any subscribers to a publication, all the updates of that publication is dropped at the edge status router. Currently the data plane uses UDP as the underlying network protocol to move data between publishers, status routers and subscribers.

2.1.2.1 Multicast

GridStat uses the multicast technique to send the same information only once over the same link. This means that a status update that has more than one subscriber is routed as a single update as far as possible before being split up into separate updates.

Figure 2.4 illustrates how multicast can be used to reduce the bandwidth needed to support two subscribers to the same publication. Both *Sub-1* and *Sub-2* subscribe to the publisher in cloud A with different rates; *Sub-1* subscribes with the rate of 1 while *Sub-2* gets an update with the rate of 2. This means that every other time a status update is routed to *Sub-2*, *Sub-1* also needs one. Instead of sending a separate copy of the status update to Sub-1 a single update is routed as far as possible before being spilt up into two copies. In this case the single copy of the update is routed all the way from the publisher in cloud A to the middle of cloud B before being duplicated into two streams.

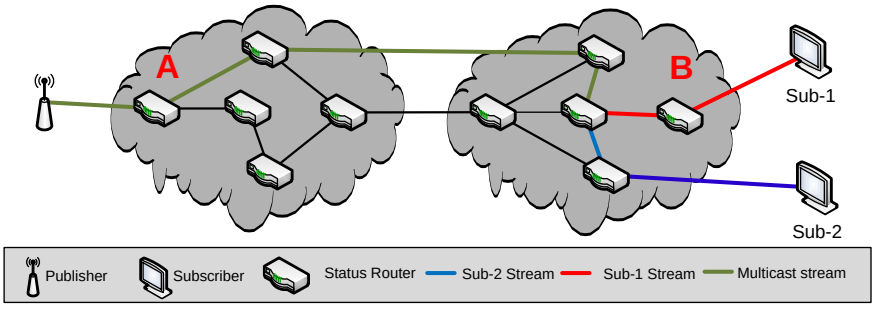

Figure 2.4: Multicast

While the use of multicast decreases the bandwidth it places some restrictions on the development of security architecture since it excludes the use of any technique that differentiates the status updates based on the receiver. For multicast to work the update that is sent needs to be identical regardless of the intended receiver.

2.1.2.2 Redundant Paths

GridStat employs redundant paths between publishers and subscribers that are set up by the management plane to achieve a tunable levels of availability. Figure 2.5 illustrates how more than one path can be allocated to achieve spatial redundancy. The primary path is colored blue while the redundant path is red. Notice that it is not always the case that the shortest path is the fastest in overlay networks.

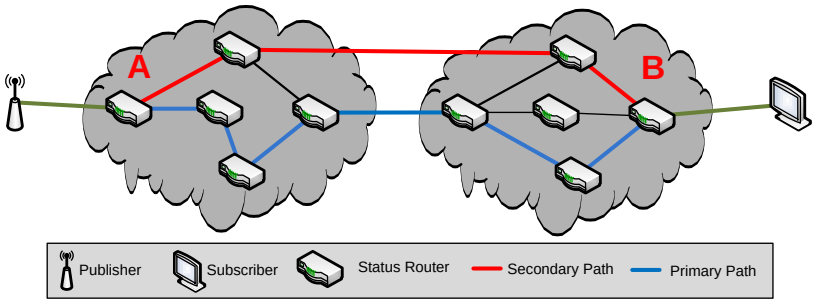

Figure 2.5: Redundant Paths

While redundant paths cannot improve the worst-case latency it can, in cases with network faults, reduce the average latency and provide an increased probability for the status update to reach the subscriber. If one of the distinct paths encounters difficulties that either slows down the delivery, or in the worst case drops, the status update, there is still a chance that the second path can deliver it.

2.2 Conventional Power Grid Information Security

The power industry has until recently not focused on security for their information systems. The protocols they have developed have been focused on increasing the power system reliability, while little thought has been put into the security aspect. *Security by obscurity* was thought to be sufficient. As is stated in [15] the power industry generally thought:

> *Who could possibly care about the megawatts on a line, or have the knowledge of how to read the idiosyncratic bits and bytes appropriate for one-out-of-a-hundred communication protocols? And why would anyone want to disrupt power systems?*

In the advent of the changes to the world situation, the increasing reliance on the information system and the standardization of protocols, this attitude has slowly changed. In the late 1990's the The International Electrotechnical Commission (IEC) Technical Council (TC) 57 Power Systems

Management And Associated Information Exchange, which is responsible for developing international standards for power grid information systems, created a working group called WG15 to explore the security aspects of their protocols.

2.2.1 IEC TC57 WG15

WG15 is an IEC TC57 working group with the title *Power system control and associated communications - Data and communication security*. Since its creation in 1997 it has tried to develop security mechanisms for the power grid information system. It has defined four main types of desired security properties, confidentiality, integrity, availability and non-repudiation and explored how to provide safeguards against them. Figure 2.6 show the types of attacks the group envisions and which types of attacks they actively try to address [15].

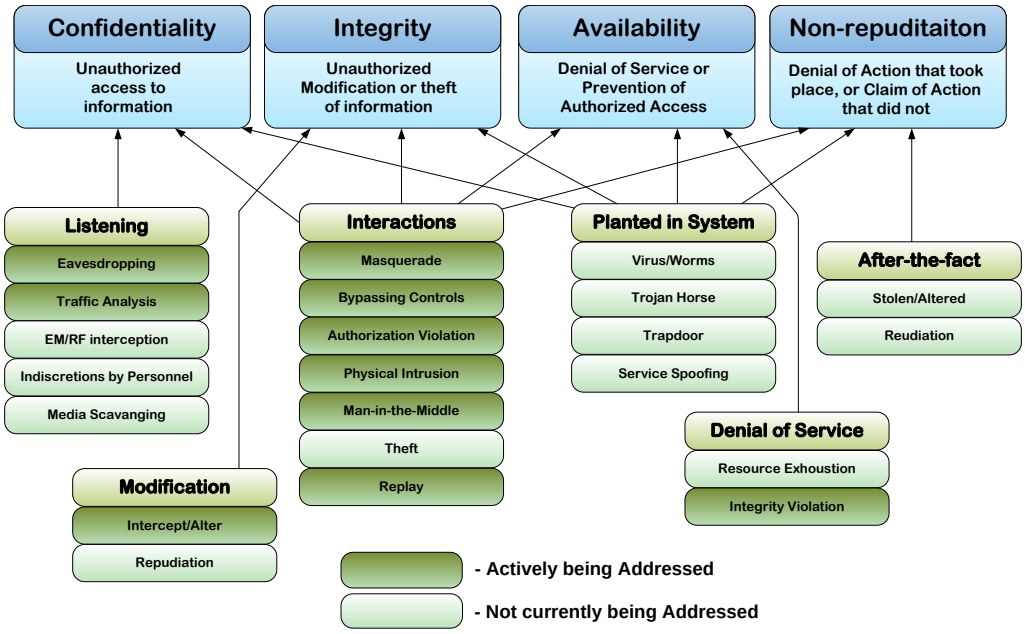

Figure 2.6: Attacks defined by IEC TC57 WG15

WG15's ongoing work follows a complementing approach instead of new development. They focus on retrofitting existing power grid protocols presented in Table 2.1 with well known general security techniques instead of developing new systems with specialized security integrated from the

IEC 60870-5	Used in Europe and other non-US countries for SCADA system to RTU data communications
DNP3	Derived from IEC 60870-5 and is in use in the US and now is widely used in many other countries as well, primarily for SCADA system to RTU data communications
IEC 60870-6 (ICCP)	Used for communication between control centers and between SCADA systems
IEC 61850 (GOOSE)	Used for protective relaying, substation automation, distribution automation, power quality, distributed energy resources, substation to control center, and other power industry operational functions.

Table 2.1: Protocols supported by WG15

Security Protocol	**Domain**	**Techniques employed**
IEC 62351-3	Profiles That Include TCP/IP	Security Certificates (PKI) and TLS
IEC 62351-4	Profiles That Include MMS	TLS and Association Control Service Element (ACSE)
IEC 62351-5	Serial Communication Profiles	Simple authentication
IEC 62351-5	Peer-to-Peer Profiles	Simple authentication

Table 2.2: Security protocols developed by WG15

beginning. WG15 has presented proposed security protocols for each of these protocols, primarily based upon the use of *Secure Sockets Layer* (SSL) and its successor *Transport Layer Security* (TLS) where it is computationally possible as shown in Table 2.2. In cases where the lack of computational power makes the use of SSL add too much latency, such as with IEC 61850 that has a 4 millisecond end-to-end latency requirement [15], they do not employ any techniques to achieve confidentiality only authentication.

By only considering the use of general security techniques developed for the general Internet such as TLS, WG15 is unable to provide the performance needed to achieve confidentiality when the latency requirements for the data is tight. They also lock their components to a static security mechanism that needs manual update to handle changes that might arise during its life time. There

is obviously a need for a more dynamic and specialized security architecture that can provide confidentiality to all types of information even with tight latency requirements.

2.3 Public Key Infrastructure (PKI) based security

Since the introduction of public key cryptography in 1976 [17] and the development of RSA in 1978 [33] *public key infrastructure*, or PKI, has grown into an integral and crucial part of modern life. In fact, it can be argued that the rise of Internet commerce wouldn't have been possible without it. PKI architectures provide the security for everything from online banking and shopping to online voting.

Generally public key cryptography, or asymmetric cryptography, is built upon the use of a pair of related keys, a public key and a private key. One key is used to encrypt while the other is used to decrypt. This means that if each party has one of the keys they can communicate securely without knowing the other key.

Since public key cryptography is asymmetric, a message encrypted with one key needs to be decrypted by the other, it can be used either to achieve confidentiality or authentication. If the public key is used to encrypt the message, only a node that has the private key can decrypt it, effectively creating a way for many-to-one one-way confidential communication channel. If the private key is used to encrypt the message every node with the public key can decrypt it which means the message is not confidential, but since only the private key could have encrypted the message in the first place, the origin of the message can be authenticated. This is often called *electronic signatures* [14].

Asymmetric algorithms generally need much longer keys than symmetric algorithms to achieve the same level of security since the relationship between the keys in a key pair can be exploited to quicker derive the key by brute force. Microsoft recommends the use of 4096 bit keys for root-certificates[2] which places huge restrictions on what problem domains it can be used. Many problem domains do not have the processing power or the time to employ such large keys.

Public key infrastructures build on public key cryptography, but only use it to achieve trust and to agree upon a faster and less resource greedy symmetrical session key for the real data transference. This way the performance hit of using asymmetric algorithms can be partially mitigated. Section 2.3.1 gives a brief overview of the the ruling PKI standard ITU-T X.509, while Section 2.3.2 explores TLS which is a widely used partial implementations of X.509.

2.3.1 X.509

X.509 is being developed by the Public-Key Infrastructure working group (PKIX) and was first proposed in 1988. It has gone through two major updates since then, one in 1993 and one in 2005 [3]. X.509 specifies standards for formats, *certificates*, certificate validation and a hierarchical composition of certificate authorities (CAs).

Certificates combine public keys with digital signatures and something that identifies them, e.g. an IP address. These certificates are sent from the server side to the client side (called an *end node* in X.509 terminology) of a connection so that the clients can authenticate the server by ascertaining that the signatures of the certificates are valid. Figure 2.7 illustrates how public key cryptography can be used to accomplish this. If the signature is valid the client can conclude that the public key it received is the correct key for the server with the specified name and thus assume that the server is the only one that can decrypt messages encrypted with the public key.

There are two ways to sign a certificate, either it can be *self-signed* which means that the server signs his own certificate with its own key before sending it to the client. Self-signed certificates achieve little security when sent over-the-wire. The only thing a client can conclude from such a certificate is that whoever sent the certificate possesses the private key it was signed with. For self-signed certificates to provide any security they have to be loaded out of band from a trusted source [3].

The alternative way is to use trusted third parties, that in X.509 terminology are called *Certificate Authorities*, to sign the certificate. By signing a certificate the CA endorses the server and

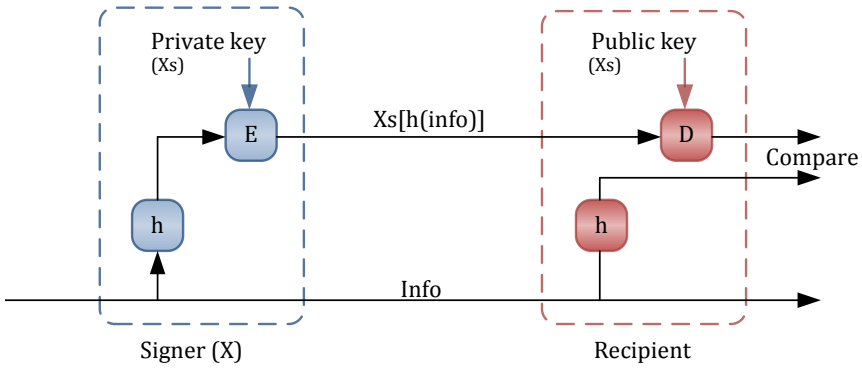

Figure 2.7: Certificate signature validation

says: If you trust me, you can trust that he is who he says he is. This assumes that the client already got the CA's public key installed and can use it to validate its signature.

The use of CAs creates the possibility of a hierarchy of CAs where each level endorses the one below and one root-CA on the top uses a self-signed root certificate. As RFC 4210 [11] specifies, this root certificate must be loaded out of band into the end node and cannot be securely sent over the wire. In practical implementations this is usually accomplished by supplying lists of root certificates that are installed with operating systems and browsers, but how the root certificate should be supplied is not defined by X.509.

A certificate may be revoked if it is discovered that its related private key has been compromised, or if the relationship (between an entity and a public key) embedded in the certificate is discovered to be incorrect or has changed. X.509 does this by checking if a certificate is valid through the use of a certificate revocation list (CRL) whose address is specified in the certificate. A X.509 certificate roughly contains the following information:

- The public key being signed.
- A name, which can refer to a person, a computer or an organization.
- A validity period.

- Certificate Authority identification.
- The location (URL) of a revocation center.
- Name of the algorithm to use.
- The digital signature of the certificate produced by the CA's private key.

X.509 also defines an optional entity, called *Registration Authority* (RA), that complements the CAs by taking care of personal authentication, token distribution, revocation reporting, name assignment, key generation and archival of key pairs. Figure 2.8 illustrates how the different X.509 entities interoperate and are organized.

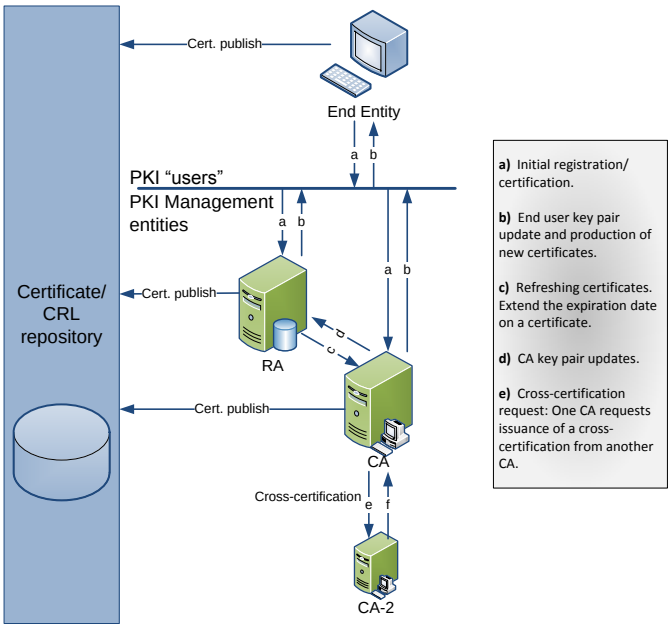

Figure 2.8: The X.509 Architecture

2.3.1.1 Weaknesses of the PKI architecture

Even though PKI has been embraced for Internet security, it is not a silver bullet for every type of security problem. As [19] explores, there are some risks associated with the use of PKI. Most of

the risks are related to the PKI implementation and use on the general Internet where trust issues are raised such as whether end users really can thrust CAs, but there are also some limitations inherent to the PKI architecture.

PKI security is built upon the assumption that there exists a secure root certificate and, since root certificates must be loaded out of band, they cannot easily be updated once installed. Even root certificates with the recommended 4096 bit sized keys do not have eternal lifecycles and needs to be updated. Microsoft recommends replacing a standalone root certificate every 20 years [2]. This is more than long enough for this to be a non issue for conventional Internet use since users replace their browsers and operating systems at a much faster rate than 20 years, but it poses a problem for systems where out of band updates do not have the same rate of update. Information systems for the power grid, on the other hand, are highly distributed with innumerable unmanned devices and have a life expectancy well above the recommended root certificate life cycle. There is also no way of guaranteeing that the key is not compromised through unforseen circumstances, such as through leaks, whether they are caused by human error or discontent employees. This further emphasizes the need a security architecture that does not rely on only out-of-band replaceable system wide keys.

2.3.2 Transport Layer Security (TLS)

Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL) are X.509 based protocols that provide a way to set up end-to-end secure communication channels between two secure nodes over a unsecured network. TLS uses the latest X.509v3 standard [11] for its certificates and is widely used to achieve confidentiality, integrity and authentication in areas such as web browsing, e-mail and instant messaging.

The TLS protocol is divided into four phases. As Figure 2.9 illustrates the first phase lets the server and client initiate communication by exchanging random numbers. In the second phase the client receives the server's certificate and verifies it while the server in phase three receives the

clients certificate encrypted with the client's private key, and a pre-master-secret encrypted with the servers public key, which is used to calculate a master secret that phase four can use as a symmetric key[16].

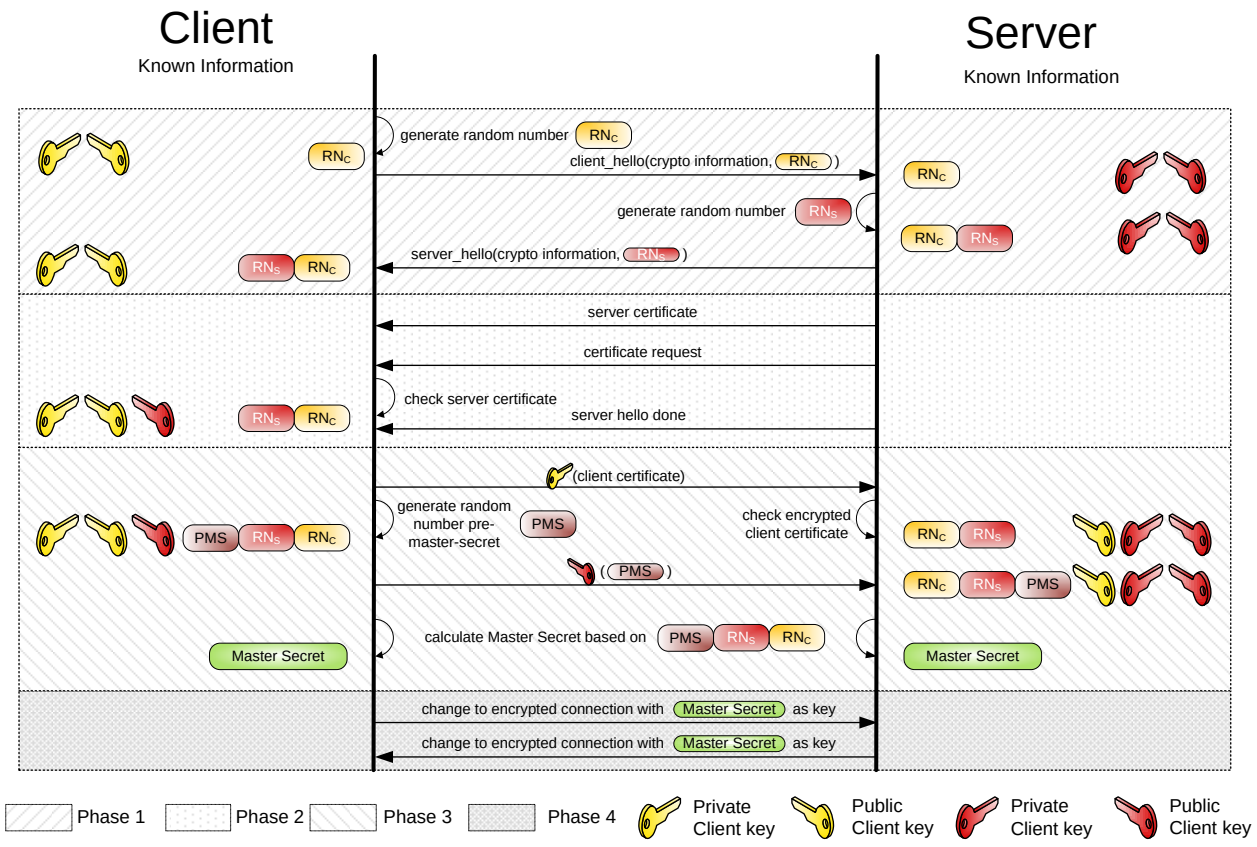

Figure 2.9: Transport Layer Security (TLS) protocol

Several different security algorithms are supported by TLS. For authentication *RSA* [33] and *Digital Signature Algorithm* (DSA) [4] are used. While RSA is used both for encryption and signing, DSA is a digital signature algorithm only. Either MD5 [31] or SHA [5] is used to for integrity, while IDEA, RC2, RC4, DES and 3DES are supported for the symmetric encryption. Some of the attributes of the supported symmetric encryption algorithms are presented in Table 2.3.

As explored in Section 2.2.1, the power industry has proposed to use TLS as the main security

Algorithm	Cipher Type	Block Size	Key Size	Ref
IDEA	Block	64	128	[25]
RC2	Block	64	40	[32]
RC4	Stream	N/A	40 and 128	[23]
DES	Block	64	64(56[a])	[6]
3DES	Block	64	168(128[a])	[7]

Table 2.3: Symmetric algorithms supported by TLS

[a]Effective key size

protocol for their information system. Since TLS follows the X.509 protocol it inherits its strengths and weaknesses and this thesis will theretofore argue that it is not the right protocol for GridStat and present an alternative security architecture.

2.4 Publish-Subscribe Security

The publish-subscribe research tends to focus on performance, scalability and expressiveness [36] and the research on security related matters that is being done is mainly concentrated on *content-based* publish subscribe (CBPS)[29, 28, 30, 24]. In CBPS systems the subscribers register filter functions and events are routed based on evaluations of these functions applied on the, usually complexly structured, events. Since the brokers in a CBPS require full or partial knowledge of the content of the events to route correctly the introduction of confidentiality poses interesting research questions. Most approaches published on how to address these requirements have been different variations on encrypting each attribute of an event with a different key, thus allowing brokers with different needs to decrypt just what is needed to route the event successfully [28]. Khurana [24] presents an extension to this approach that supports events that have an XML structure. Others employ end-to-end techniques such as only encrypting the values of events, while keeping the attribute types in clear text to route the information [30].

While most research on security in CBPS is not directly applicable to rate-based status dissemination publish-subscribe some of it can be useful such as Wang et. al. [36] which does not present a security architecture, but explores the issues related to security in the publish-subscribe

paradigm that can be used as part of the threat model specified in Section 3.1. Pesonon et. al. [29] present a security architecture for multi-domain CBPSs that acknowledges the fact that confidentiality in multi-domains warrant special considerations such as to what the intermediate brokers are allowed to know about the content of routed events and where the access control to the events should be placed. The architecture employs two separate security schemes, one for the connection between the end points (publishers and subscribers) to the brokers based on TLS and another for the inter-broker communication that allows the brokers to access the events partially.

A common trait of the security architectures published for CBPS is that they all employ PKI in a lesser or greater fashion. This introduces weaknesses that are introduced in Section 2.3 and further explored in 3.3. They also use asymmetric algorithms such as RSA to achieve event/status update authentication, which is well known for adding unacceptable levels of latency to real-time status dissemination and confirmed by the performance evaluation in Section 5.2.1, and a simple alternative will be presented in Section 4.6.4.

CHAPTER THREE

DATA PLANE SECURITY ARCHITECTURE

In this Chapter the general architecture behind the data plane security system is presented. Introductory the threat model developed based on threats to information systems for the power grid and generally desired security properties for publish-subscribe systems, is presented.

Then the idea of building a security architecture around the use of transparent interchangeable software security modules to address these threats is explored. Finally, the infrastructure security needed to support such a dynamic approach is discussed. All together this provide the groundwork for the design and implementation solutions presented in Chapter 4.

3.1 Threat Model

The stated goal of this thesis is to present a security architecture that adds confidentiality, integrity and, to a lesser degree, availability to GridStat's data plane. This Section will present a convergence of two different approaches to threat models, one based on the functionality of publish-subscribe systems as defined by Wang et. al. [36] and one based on types of attacks on power grid information systems developed by IEC TC57 WG15 [15]

The Wang et. al. approach to threat models is requirements based, which means that general goals of a security architecture in a publish-subscribe network are used as a starting point. He divides the requirements into two sub categories, *application security* and *infrastructure security*. Application security comprises the security of the data flow from publisher to subscriber, while infrastructure security consists of the security of management communication. As previously observed, the main focus of this thesis is on the application security and not present a general infrastructure security architecture; only a partial infrastructure security architecture that can support the application security mechanisms. Wang et. al. further define the following application security issues for publish-subscribe systems:

Authentication: Authentication is needed for subscribers to assert that the events they receive actually originated at the correct publisher.

Information integrity: Subscribers needs to be able to check the integrity of received events.

Information confidentiality: Keeping events flowing from publishers to subscribers confidential.

While all of the application security issues above will be addressed by the security architecture presented in this thesis, the following infrastructure issues will only be partially addressed:

Publication confidentiality: Assuring that only authorized subscribers gain access to publications.

Subscription Integrity: Protect subscriptions from unauthorized modifications.

Service integrity: Protect against compromised infrastructure level components.

User anonymity: Keep the identity of users confidential.

Subscription confidentiality: Keep what the subscriber subscribes to confidential.

Availability Reduce the risk of malicious publications and subscriptions that can be used to overload the system.

Accountability: Require subscribers to be accountable for the information they receive.

These issues are based on general publish-subscribe systems and does not take into account the special needs of information systems for the power grid. To address both the publish-subscribe aspects and the power grid aspect, these issues can mapped to the threat model developed by the security group within the IEC TC57 presented in Section 2.2.1 as shown in figure 3.1. User anonymity is the only issue that cannot be mapped to a type of attack since user anonymity is

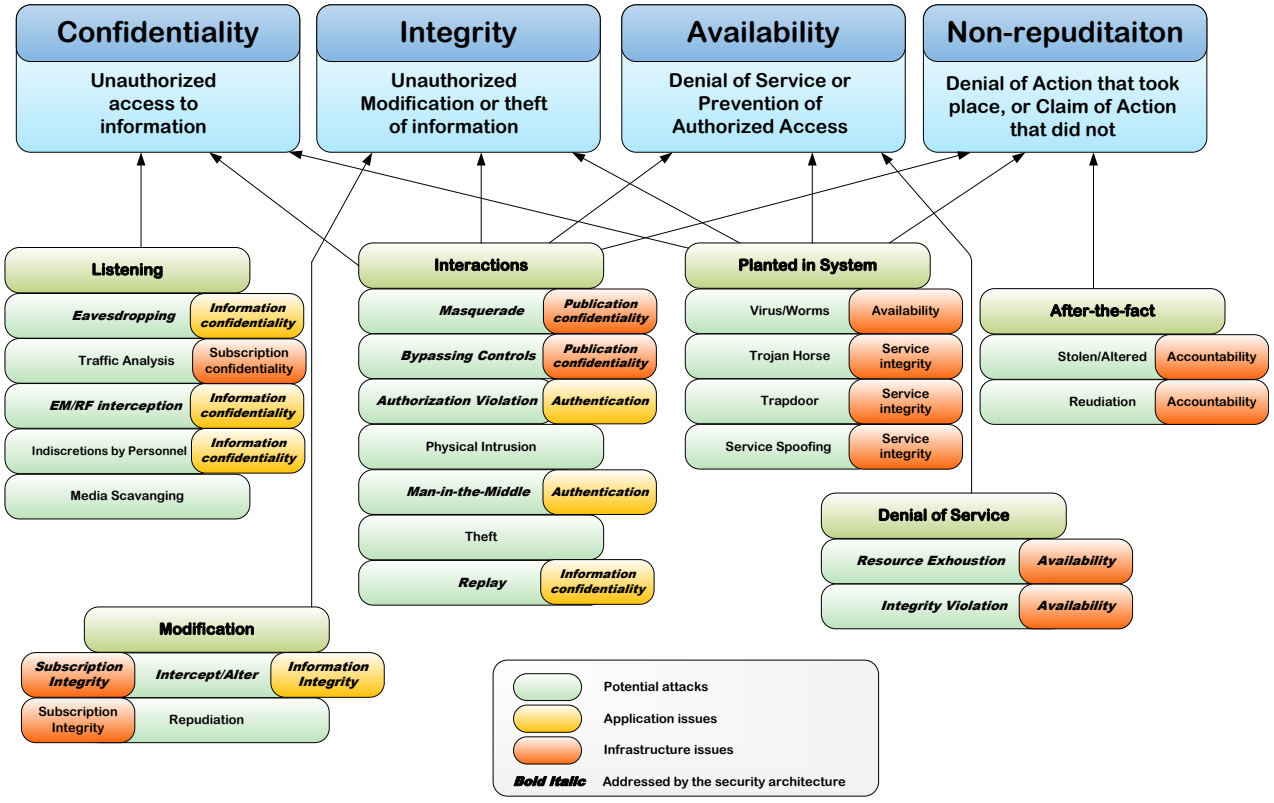

Figure 3.1: Threat Model

something to avoid in information systems for the power grid where accountability is much more important.

As the bold-italic text in Figure 3.1 illustrates, the security architecture presented in this thesis addresses only parts of the overall security question associated with an information system for the power grid, but while having a narrower focus be able to provide more deeper protection. The presented architecture will concentrate on providing information confidentiality, integrity and authentication while defending against listening and modification of data and malicious interactions with the security infrastructure in addition to partial denial of service protection.

The security of GridStat's original infrastructure, such as end-point security and QoS broker communication, is assumed to be secured by other means, or by future extension to the presented security architecture. Some fault tolerance mechanisms, such as the use of redundant paths and

backup services that already are part of the GridStat framework, provide some protection against attacks on availability, while future work needs to be done to design new mechanisms to protect against other type of attacks on non-repudiation and physical attacks on the infrastructure. Even though it is assumed that other security measures cover the general GridStat infrastructure it is not assumed that they cannot be compromised. The security architecture presented in this thesis will take into account the effect of different component being compromised and try to minimize the negative effects through the use of compartmentalizing techniques.

GridStat is primarily designed to run on a private network separated from the public Internet either physically or logically in order to achieve reliable QoS. While this reduces the chance of arbitrary attacks it does not remove the risk of more organized attacks with resources that could give them access to the private network. Therefore all considerations specified above have to be taken into account.

3.2 Interchangeable Transparent Modules

The data plane security architecture is built upon the idea of using transparent interchangeable security modules to achieve security for GridStat's data plane. This implemented as a security extension to the existing management plane, called the *security management plane*, that generates keys and assigns sets of modules from a module repository, to the publishers and subscribers, on a per status variable granularity according to dynamic policies as illustrated in Figure 3.2.

Assigning sets of modules and keys on a per status variable granularity enables the security system to address the different needs of different multicast streams with status updates. For some of these 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 use faster, but weaker, security modules. These assignments are specified in *publication policies* and corresponding *subscription policies*, which design is specified in Section 4.4.1 and 4.4.2.

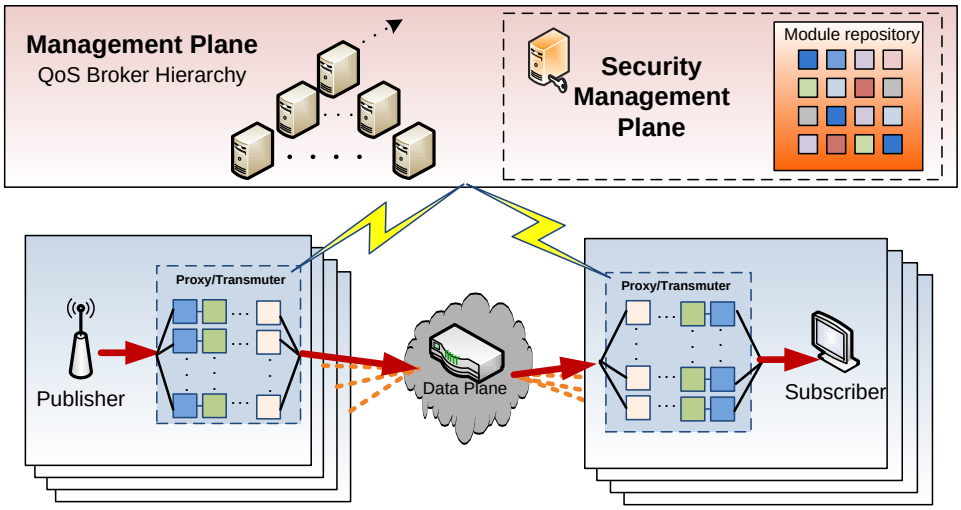

Figure 3.2: The Interchangeable Transparent Modules approach

Each publisher and subscriber receives one publication or subscription policy for each of their respective publications and subscriptions. Based on these policies they download the security modules they need from the security management plane and instantiate them with the keys specified in the policy. The keys are random generated for each of the assigned modules by the security management plane. When the modules are installed they are transparently applied to each of the publishers' and subscribers' event streams.

Above publication and subscription policies there is another layer of policies called *security group policies* (SGP) that define policies for groups of publications to ease the operation of policies. Whenever a publisher wants to start a new publication it has to register the publication with a pre-existing security group. The security management plane uses this association to assign modules to the publication policy based on the current policy for the specified security group and thus, by implication, also to all corresponding subscription policies. Any changes instituted to the SGP will be propagated down the publication and subscription policies that are associated with it in one of two ways specified in Section 4.5.

The security management plane's repository of modules' can be changed over time by adding

new modules at runtime. These new modules can both be assigned to new publications and to existing publications by adding and/or replacing old modules assigned to the different security groups. All modules added to the module repository need two have two separate parts; a file containing the actual code which will do the modules work and a *module policy* specifying the properties and behavior of the module, and one optional *key generator*, if the module needs a special type of key generations not supported by the default key generator.

The strength of a modular approach is that new security modules can be implemented with varied functionality and performance attributes. By combining the modules in different combinations the security architecture provides a unique toolset for easily making and enforcing tradeoffs between different security and performance properties on an extremely small granularity. It also enables system administrators to respond to changes in the security field by introducing new modules to replace old ones whenever needed.

There are almost an infinite number of modules that could be implemented and deployed, however this thesis defines five major groups of module types that are of initial interest, each of which has a clear and differentiated goal from the others.

Encryption modules Modules that encrypt information to achieve confidentiality.

Authentication modules Modules that with the use of digital signatures let receivers of information authenticate its origin.

Integrity modules Error check and error correcting modules whose goal is to assert the integrity of the information or correct integrity faults.

Obfuscation modules Modules whose goal is to mask recognizable patterns of data such as repeating bit sequences, which could be used to break the confidentiality achieved by the encryption module.

Filtering modules Modules that try to reduce the risk of denial of service by filtering published events so only the events that are needed are pushed into the data plane.

More information on the design of the specific proof-of-concept modules can be found in Section 4.6 and an performance evaluation of these can be found in Section 5.1.

3.3 Security in the Security Management Communication

Adding security measures to provide data plane security, while at the same time introducing new security weaknesses in the management plane on which it relies is futile. Any security system needs to protect its own management communication by providing confidentiality, integrity, authentication and availability for itself in the same way as it provides its payload, in this case the flow of status updates from publishers to subscribers in the data plane.

The challenges to secure the security management communication are many, but slightly differentiated from the challenges to securing the data plane. Communication in the data plane is based on publishers that push their information through multicast streams to multiple subscribers with real time latency requirements as described in Section 2.1.2. Through these channels a steady stream of information is transferred, which given enough time, enables an attacker to gather huge amounts of data about its security measures.

Security management communication, on the other hand, is point-to-point communication with sporadic burst of information with relatively loose latency requirements. This gives security management communication fewer restrictions on the type of security measures that can be taken than with the data plane communication.

Figure 3.3 depicts the two types of security management plane communication that needs to be secured; the communication from the data plane nodes to the security management plane represented by a *Security Management Server* (SMS) and the internal security management plane communication between SMSs. More on the security management structure and components can be found in Section 4.1.

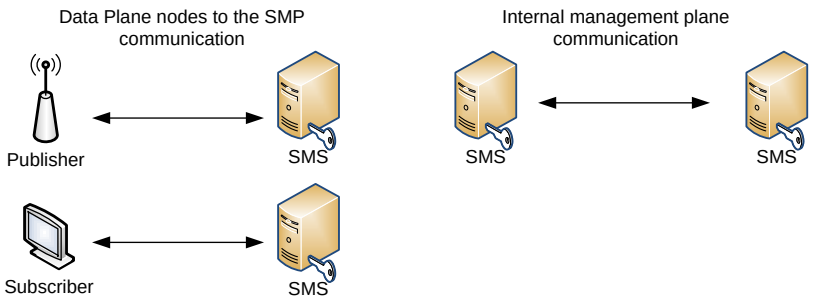

Figure 3.3: Types of Security Management Communication

The modular approach to securing the data plane introduced in Section 3.2 necessitates a secure distribution of policies and modules from the security management plane down to the publishers and subscribers. To support such a dynamic solution to data plane security a dynamic approach is also needed for the security management communication. A chain is never stronger than the weakest link and building a dynamic security system on top of a static management security would avail little. If some part of the static security is compromised, no level of flexibility in the rest of the system would make any difference.

3.3.1 PKI

Dynamic data plane security hinges on nodes in the data plane being able to securely contact the security management extension and download the modules and the keys they need. The challenge arises when the communication link is suspected of being compromised and there thus is a need to re-establish security by replacing the keys and/or modules used to secure it. As Section 2.3 explores, the conventional way of securing such communication is the use of PKI systems such as X.509. X.509 and other PKIs build upon the idea of using certificates signed by certificate authorities and asymmetrical encryption algorithms to establish secure connections. Clients are provided a root certificate containing a public key out of band that can authenticate that incoming information is signed by the corresponding private key. This provides the basis on which other layers of security can be built such as letting other servers publish their own certificate with their

own public key signed by the original root private key.

In GridStat's case employing PKI would enable each SMS to replace public keys and modules used to secure the communication to its children nodes that could be, or already are, compromised securely by signing them with the root private key, as many times as is needed. Figure 3.4 illustrates how such a replacement protocol, which replaces the current module and keys used for security, could look. While this protocol could be further extended with a second key switch to ensure that after-the-fact attacks extracting the key from its previous use with the old module to compromise the communication with the new would not be possible, the main problem here is the Achilles heel of all PKI systems, namely their inability to replace the underlying root key and authentication algorithm.

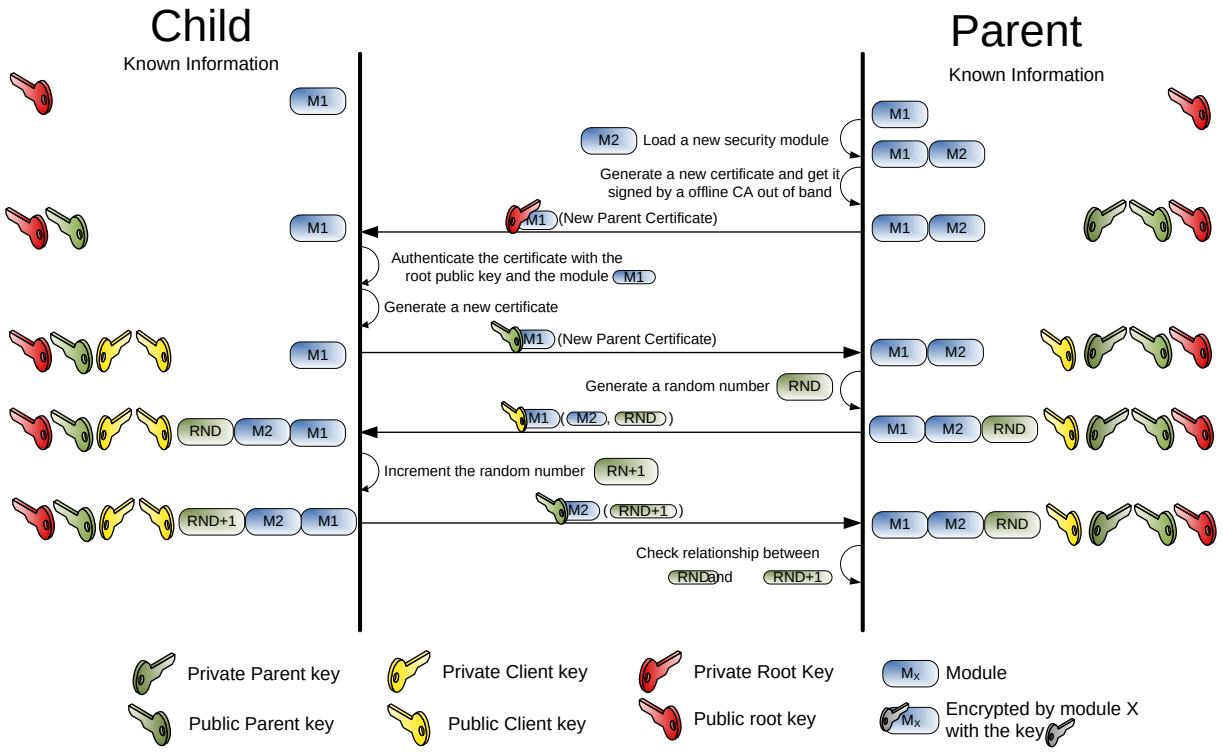

Figure 3.4: PKI module and key management communication replacement protocol

As previously stated the problem with PKI approach is that everything hinges on the integrity of

the root private key. Different PKIs have different designs, some with highly advanced hierarchies of certificates such as X.509, but they all have in common that there must exist a root certificate signed by a private key to authenticate them. If that private key is compromised the whole security system would be compromised.

Designing a security system using a dynamic and modular approach necessitates that its management communication also shares these properties. If the root private key somehow becomes compromised, no new modules or keys could be distributed safely to the data plane, a system is only as dynamic as its least dynamic component.

Alternatively, it is possible to exploit the relatively static topology of GridStat. While PKI is designed for being able to handle an ever changing field of new web services, servers and clients, GridStat is a much more stable environment. A GridStat node, being a data plane node or a SMS has only a single parent it needs to communicate with and that parent will extremely rarely be changed. In GridStat new children nodes can be added, but never without explicitly being registered at its connection point by an authorized operator. All new children need to be registered with their parent in order to be able to access its services. This means that security management communication does not need to handle dynamic connections which simplifies the problem space greatly and allows the approach pre-loading of a finite set of keys and an initial module.

3.3.2 Key set pre-loading

By providing all new nodes that are being added to GridStat and their parents with a set of k keys and an initial encryption module as shown in Figure 3.5, it is possible to build a dynamic solution with some limitations, but which removes the need for static root keys that potentially could be compromised either as a result of brute force attacks or human error.

Pre-loading k keys makes it possible to switch keys k times even if the current key is compromised. This is possible since there is no need to send any keys over the wire, something that

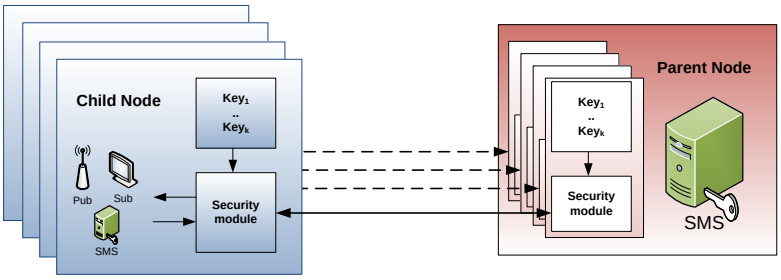

Figure 3.5: The pre-loading architecture

makes it impossible for an attacker gain access to the new key through sniffing. The only mischief an attacker can try is to provoke a situation that forces the child and parent node out of key-synchronization. That can either be attempted by initiating false key switches or interrupting valid ongoing switches.

Figure 3.6 illustrates how the parent node initiates a key switch by sending a key-switch command together with a random number encoded with the current key. The child node decrypts the number with the current key, increments the number by one, and returns it together with a new random number encrypted with the next key in the key list. This concludes phase one. If it could be assumed that the child never would receive false key change commands from attackers masquerading as the parent aimed to push the child out of key synchronization with the real parent, the protocol could have finished here. But since this cannot be assumed, the parent node initiates phase two of the protocol which asserts that both sides completed the key change successfully by first checking that the child node correctly incremented the first random number. Then the parent decrypts, increments and re-encrypts the second random number and sends it back to the child. The child checks that the random number is incremented correctly, increments it a second time and sends it back to the parent. The client now assumes that the key switch is complete and moves permanently to the new key. The server assumes the same when it receives the random number incremented for a second time.

Forcing both parties in the key switch to prove their possession of the next key in the list makes

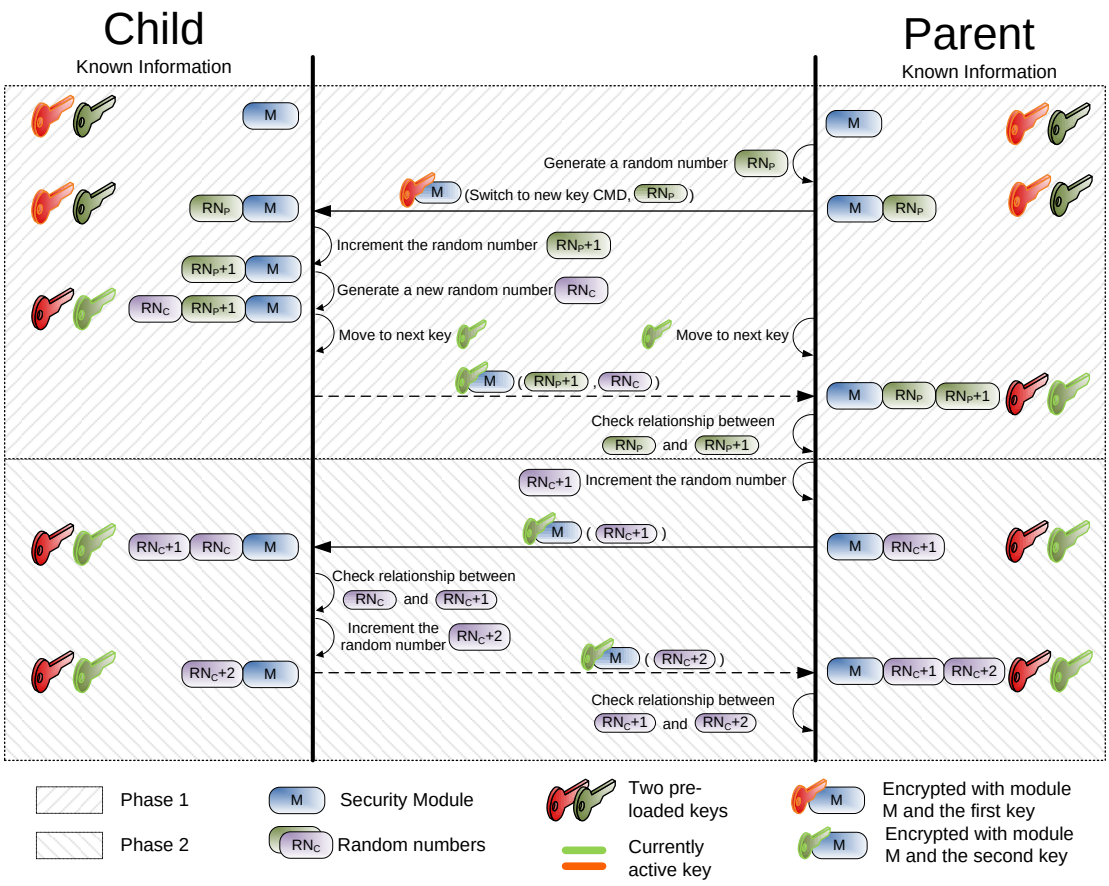

Figure 3.6: Key switch protocol using preloaded keys

it impossible for an attacker to initiate invalid key switches without possessing the next key in the list. If either of the random number checks fails, or either the first message or its reply isn't received, both the child and the parent revert back to the old key after a short timeout.

To combat the chance that interruptions or loss of the last reply from the child to the parent causing the nodes to go out of key sync, the last message has to be treated differently than the others. Since the child assumes the key switch was successful when it returns the second random number incremented by 2 without checking whether the parent node received it, the parent cannot follow the pattern of the other messages and reset back to the old key if it does not receive it. The parent now has to decide whether the child node has completed the key switch or not.

If the child did not receive the last message from the parent it will revert to the old key after a given timeout. In an effort to avoid this happening the parent first tries to repeat the last message until he gets a correct response, in which case the key change has been completed successfully, or the time before the child will revert to the old key runs out.

If the parent has not received any responses, or only invalid ones, when the timeout kicks in, nothing can be asserted about which key the child node uses, the old or the new. To resolve this question the parent initiates a series of *next-key-probes*, as illustrated in Figure 3.7, with alternating base keys. First the parent sends a probe using the old key as base, if no reply is received, or the reply number is wrong, a new probe using the new key is sent. The parent will continue to do this until it gets a correct reply.

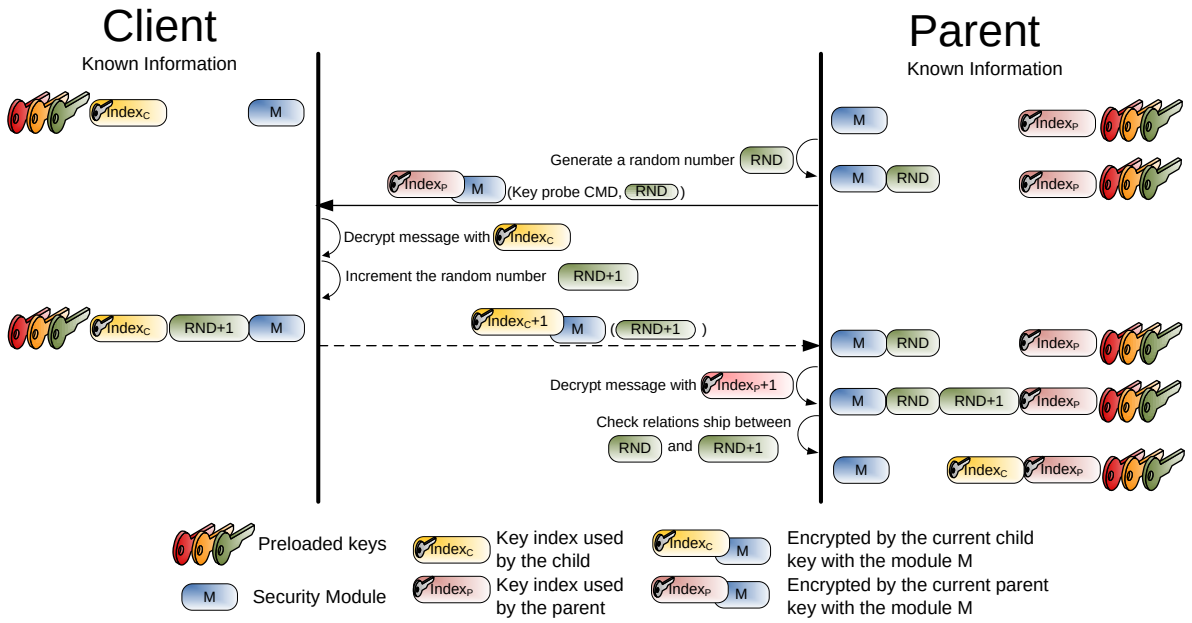

Figure 3.7: Key Re-synchronization protocol

Since the child has to use either the new key or the old key there can be only two reasons for the parent not to receive a correct response. Either there is network failure, which means that when the network is fixed the probes will re-synchronize the keys, or a man-in-the-middle keeps

intercepting the commands. In the case of the man-in-the-middle attack the attacker needs to be able to continuously intercept all the commands between the child and the parent to keep such a denial of service attack up. As soon as the interception stops, the nodes will re-synchronize. Worth noting is that if the attacker has the control needed over the network needed to accomplish such denial of service it can completely sever the communication between the two nodes without the need to attack specific protocols. Adding additional elaborate measures to combat such attacks on specific protocols would thus be fruitless.

Key switches in themselves cannot handle the case where the current key is compromised as a result of the module being compromisable. When the module is compromisable the attacker can by listening in to the message-flow extract the key used. This means that in general all messages that are sent using this encryption module, no matter the number of key switches, are insecure and open to man-in-the-middle attacks after a undeterminable amount of time.

By utilizing the fact that modules in themselves are not secret, it is possible to extend the key switch protocol to also replace the current module, even though it is compromised. Successfully replacing a compromised module with a new module necessitates the transference of the new module from the parent to the child without any men-in-the-middle compromising the modules integrity. To create a special case where the chance of such a successful man-in-the middle is well below acceptable levels the following assumptions about what preconditions an attacker needs to satisfy to extract a new key from the currently used module have be exploited:

1. Assumption: A relatively large number of processing cycles, which takes time.

2. Assumption: A significant amount of data to base their algorithms on.

These assumptions can be taken advantage of. First of all, by switching keys when sending a new encryption module the attacker is denied two things.

1. They cannot use any key they previously have decoded from the communication stream.

2. They have to discard all previously collected data about the stream and start from the beginning.

Since the amount of data needed to transfer a new module is relatively small, the chance of the attacker getting enough data from that transfer alone to decode the new key is very low. Observe that for a man-in-the-middle attack to successfully compromise the integrity of a module transfer the attacker needs to be able to replace the real module with a fake module that it has encrypted with the currently active key. By switching to a new key just before sending the module the attacker's workload can be significantly increased by forcing it to extract the new key on very little data before being able to do the encryption. If the attacker by chance should get enough information in that single message with the new module to extract the new key, it still needs some time to calculate it. By adding strict time limits on these transitions the attacker would need to accomplish all this without adding a significant level of latency.

To further enhance the protocol, keys could be switched a second time after the new module is transferred. This will ensure that when the security management communication link starts using the new module, it got a new key that has not been used with the old module. Something that, assuming the new module is secure, makes it impossible for the attacker to use information gleaned from the use of the old module with the new module. Figure 3.8 depicts the module switch protocol, red arrows symbolize messages encrypted with the old key and module, orange arrows messages encrypted with the temporary key and old module, while the green arrows symbolizes safe communication using the new key and the new module.

First the parent sends a replace module command encrypted with the old key and module. Both the parent and the child moves to a temporary key. Then the parent sends the new module and a random number, encrypted with the temporary key and the old module. When the child receives message C it decrypts the new module and random number, installs the new module, increments the random number, generates a new random number and then changes keys again to a new key.

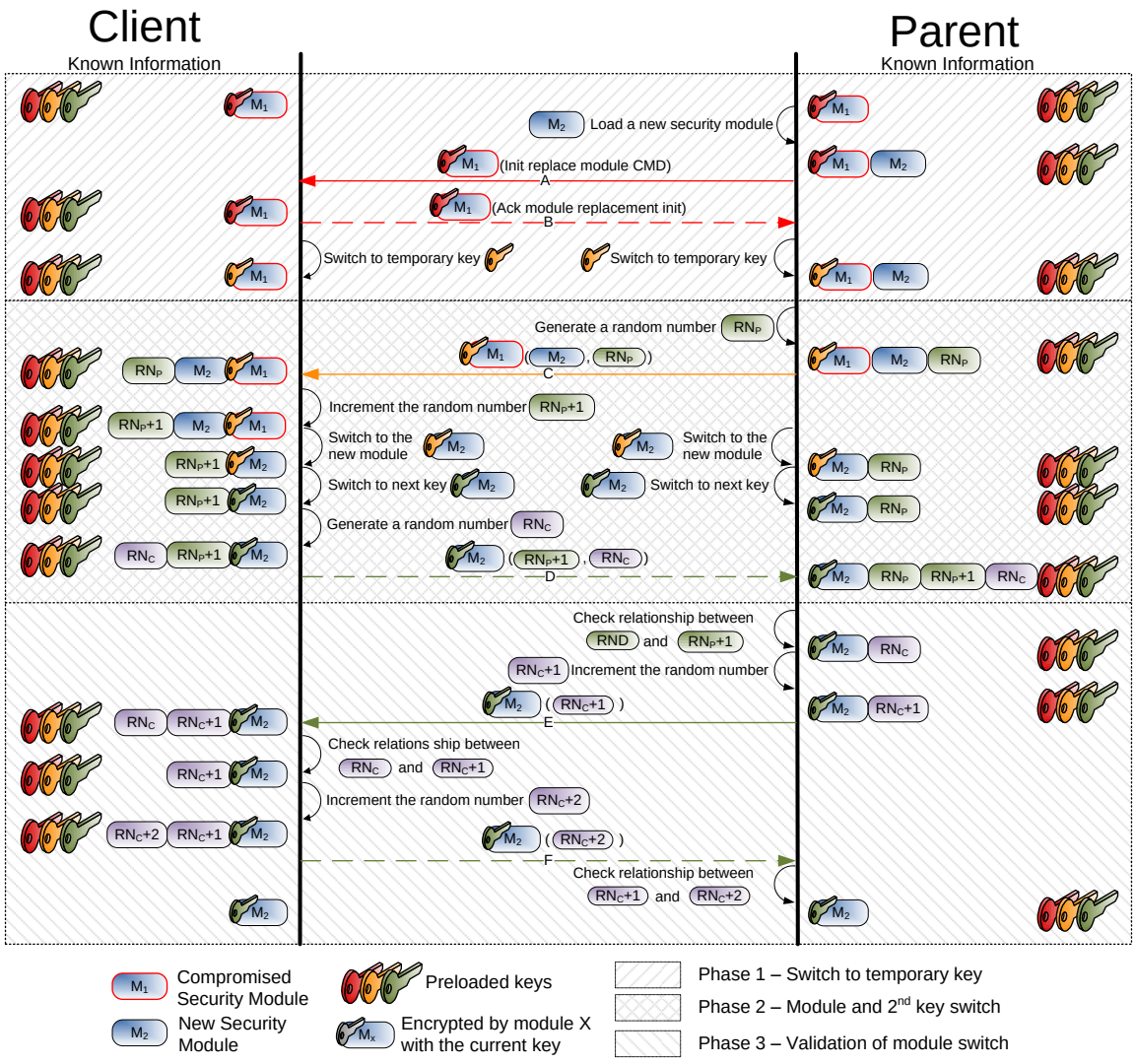

Figure 3.8: Module switch Protocol using preloaded keys

Finally the child replies to the parent with the two random numbers encrypted with the new module and the new key. Receiving message D the parent also moves from the temporary key to the next key and decrypts the message.

This module switch protocol would make it extremely hard to perform a man in the middle attack and should reduce chance of a successful attack to acceptable levels. The essential part here is to enforce a tight time requirement on a response to message C. If the parent does not receive

a message D within a set time it red-flags the node and aborts the operation. To be able to do a successful man in the middle attack the attacker must be able to extract the temporary key from the single message C in a short enough time to encrypt its false module with this key and send it to the child without exceeding the time limit.

The third phase in the protocol is added to assert that the module switch was successful and avoid the problem of loss of the last confirming message, caused by an attacker or by mundane network problems, resulting in the child and parent to go out of module and key synchronization. Loss of any intermediate messages can easily be solved by letting both parties revert to old keys and modules after a timeout, effectively re-synchronizing the parties. If the parent does not receive F, or the response is invalid, a similar approach to handling the loss of the last message in the key switch protocol illustrated in Figure 3.7 can be employed to re-synchronize the child and parent.

As shown it is possible to replace a module $k/2$ times if the child is preloaded with k keys. Even though this definitely imposes a finite limitation on the dynamic aspects of the security architecture, it can be argued that with the correct size of k this would not hamper the security significantly. The only reason for needing an infinite number of keys is that the keys and modules keep getting compromised indefinitely. For this to be true there has to be another weakness in the security system that no amount of key or module switches can remedy. Assuming a finite life expectancy there should exist a k such that there are enough keys to switch modules as many times as needed during the deployment of GridStat.

Calculating the size of k is outside the scope of this thesis, and would be strictly application dependant, but a simple example is to base the size of k on the expected deployment time divided on the time it takes to replace a module. This would result in the maximum number of keys it is possible to use during the deployment phase of the security architecture's lifecycle. The actual size of k would be much smaller, but it informally proves the point that having a finite k not necessarily makes the security architecture more static.

The pre-loaded approach provide a second level of security. While the data plane communication is secured through the use of assigned data plane security modules and keys that can be replaced without limitations, the pre-load load approach secure the transfer of these data plane modules and keys with the limitation that the modules and keys securing each of these security communication links only can be replaced a finite number of times.

3.4 Coupling of the Management Plane and the Security Management Plane

There are three major integration models that the security extension to the management plane could be use; either letting it be completely integrated into the existing management plane, partially integrated or completely decoupled. All of these have their own sets of tradeoffs.

Complete integration as shown in Figure 3.9 entails that the security management plane is integrated into the existing QoS hierarchy to the level that the security management servers only live as a logical group of functionality in the QoS Broker processes. A by-product of such an approach would be that by sharing the process, they also share the same interface which makes the security management plane completely transparent for the data plane. It also becomes transparent to include the security extension to any backup replication scheme for the management plane.

A problem with complete integration is the loss of flexibility that can become important in the future work of extending the data plane security architecture into the QoS hierarchy. By muddling what could be separate functionality into one component the complexity is increased and the ease of understanding degraded.

Letting the security servers be modules, with their own interfaces, instead of integrated logic, enables the possibility that the QoS brokers can communicate with SMSs in a similar way to what the data plane nodes are doing through the same secure interfaces. This secure communication would provide a secure base from which it is possible to provide secure hierarchical communication using the Ratatoskr RPC [35] calls. When the QoS hierarchy uses Ratatoskr RPC calls they act

like publishers and subscribers and can thus be treated as such transparently by the security management plane if the QoS brokers communicate through the security management servers' external interface.

A third tradeoff associated with a completely integrated solution is that the *security management plane* (SMP) topology by definition will be bound to the management topology. SMSs have a different utility model than QoS Brokers and thus might be suited for other flatter or deeper topologies than the QoS brokers in the different deployments.

With the partial integration approach depicted in Figure 3.9 the SMSs still live on the same hardware as the QoS brokers, but in separate processes and with separate interfaces. This reduces the level of transparency, but greatly increases the level of flexibility, both on the software engineering side by enabling simpler non-intrusive implementation, and by allowing the Ratatoskr RPC transparent access to the SMP services. It also allows for the flexibility of quickly, without the need for code changes, moving to a completely decoupled approach if that is desired.

A complete decoupling of the security management plane as shown in Figure 3.9, would further increase the flexibility of the system by enabling the security management plane to be deployed in a completely different topology from the QoS Broker hierarchy. This would make it possible to deploy the security management plane in an optimized topology for each situation, but make it necessary to provide it with its own backup replication scheme to ensure acceptable levels of fault tolerance.

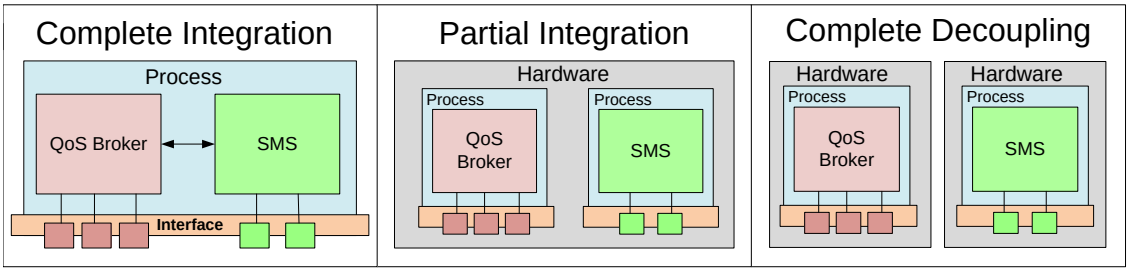

Figure 3.9: Security Management Plane Intgeration

In an effort to maximize the flexibility without adding too much complexity the presented security architecture will employ a partial integrated approach. While this adds some restrictions on the topology of the security management plane it makes it fairly easy to include the SMP into the existing backup replication scheme and can in addition quickly be moved to complete decoupling if that is desired.

CHAPTER FOUR

DESIGN

This chapter presents the design and implementation of the security architecture presented in Chapter 3. It first fleshes out the design of the security management plane and explores some tradeoffs that had to be done in Section 4.1, then moves on to the enhancements done to the data plane to support the architecture.

The chapter then proceeds to present the `Transmuter` class and its crucial mechanisms in Section 4.3 before it moves on to the five XML policy definitions developed for the security extension. Section 4.5 delves into the different policy propagation mechanisms that let modules and keys be assigned and reassigned to the publishers and subscribers.

Finally Section 4.6 defines the different types of security modules needed to provide the necessary security and presents a set of implemented proof-of-concept modules based on these types.

4.1 Security Management Plane (SMP)

As described in Section 3.2, the SMP is an extension to the existing management plane. This extension controls the security aspects of the data plane by assigning, re-assigning and distributing dynamic sets of security modules and keys to the publishers and subscribers. The simplest way for the SMP to achieve these properties would be to just let a single *Security Management Server* (SMS) control the whole data plane, but this is not a viable solution since this would add a unacceptable scaling constraint and introduce a single point of failure.

To mitigate both the scaling and single point of failure problem, the SMP needs to be divided into sets of SMSs each serving their own subset of the data plane. There are several ways to organize these SMSs into topologies, however, given the hierarchical structure of the existing management plane, the most logical is to use a form of hierarchical topology that can utilize the existing command structure and replication scheme for increased fault-tolerance as shown in Section 4.1.1.

Section 4.1.2 and 4.1.3 explores the design of leaf-SMSs and interior-SMSs respectively.

4.1.1 Hierarchy

The security management plane manifests itself as a set of security management servers organized in a hierarchy with a 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 development of an access control scheme that provide the flexibility needed for a cross domain security system, such as the one outlined here, is outside the scope of this thesis. The security management servers instead employs a simple access list scheme to decide whether a subscriber should be granted access to a publications modules and keys. However there is an ongoing project where TrustBuilder [38, 26] is used to provide attribute based incremental trust negotiation. More details on this project can be found in the future work Section 6.2.7.

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 pier-to-pier scheme of direct communication between leaf-SMS cannot support. With the pre-load scheme, outlined in Section 3.3, of all communication links between components, except the data plane communication, need to be explicitly supplied with a set of keys and a module. In a pier-to-pier approach this means that all SMSs needs to be supplied with the keys and modules needed to communicate with all other SMSs. In addition to the resources needed to store all the keys, the pier-to-pier approach would make it impractical to add new SMSs since they have to be registered

with all the other SMSs. By letting the hierarchy of interior-SMSs handle the inter leaf-SMS communication, new leaf-SMS can be added by only registering it with its parent interior-SMS, thus greatly reducing the need for key storage resources and increasing the ease of use.

To address the utilities concerns about letting others control the access to their information, all publication information, and its access control, is kept at the leaf level and only the leaf-SMS that serves the cloud of the publisher can grant access to it. This also makes it impractical to cache subscription policies in the hierarchy since only the leaf-SMS that owns them can grant access to them. In addition it is of outmost importance that the policies returned to subscribers are guaranteed to be the most recent, something that makes the use of caching of the policies cumbersome.

Modules, being much larger in size than policies, have a much greater potential benefit from being cached. They also lack the cache drawbacks of integrity problems and access control since the modules are static in nature and not sensitive data in themselves. Section 5.3 will show the average latency associated with requesting missing modules for subscribers can be greatly reduced by letting the SMS hierarchy cache the modules.

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.

The two phase approach take advantage of the fact that modules can be cached, while policies cannot, by separating these two into two different rounds of communication. In addition to ensuring that the modules are not transferred unnecessary if they already are cached further down the path, the two phases also allow publishers and subscribers to download the needed modules in parallel, significantly improving performance. Figure 4.1 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.

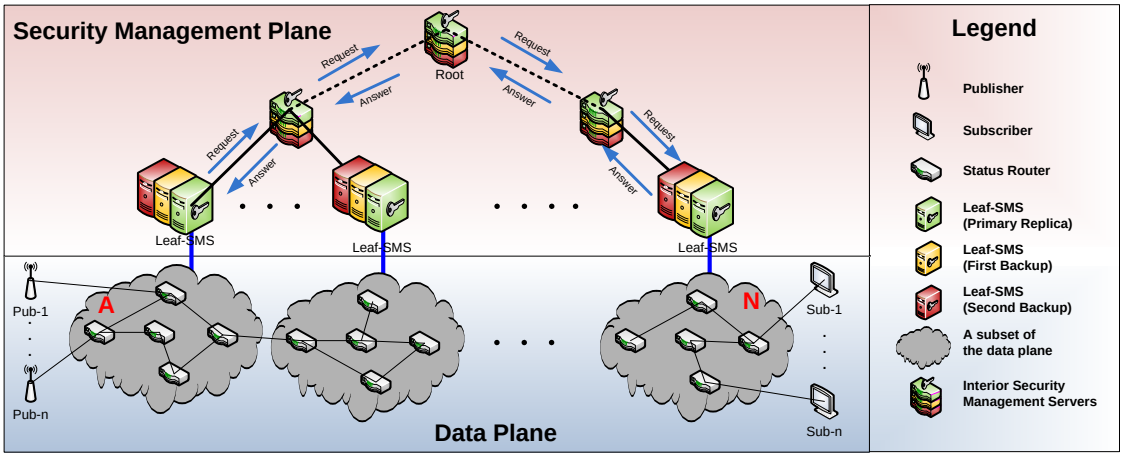

Figure 4.1: The Security Management Plane Hierarchy

To set up a 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. If the module is

cached in the hierarchy these modules will be returned to the subscriber relatively quickly, but in the worst case only the leaf-SMS with the publication information has the modules requested. In that case the modules will follow the exact same path as the policy did, but this time the modules will be cached and thus provide better performance for further subscription requests that use these modules.

Availability is an important aspect of security systems where the data that it secures is of vital importance and, even though the actual implementation of mechanisms such as backup servers and replicas is outside the scope of this thesis, the architecture is designed with such extensions in mind. By having a hierarchical structure that mimics the structure of the existing GridStat management plane, it can easily utilize the accessibility mechanisms that are, and will be, developed for it. The implementation of the leaf-SMS and interior-SMS is done in such a way that all information is stored in XML structures. This means that backup replicas relatively simply can be deployed by replicating the content of these XML databases and thus replicate the SMS's states. This fits well with the replica approach for the existing management plane and Figure 4.1 also illustrates how the GridStat replica scheme for the management plane could be extended into the SMP.

4.1.2 Leaf Security Management Server (Leaf-SMS)

The leaf-SMS serves 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 publishers in the cloud register and issues subscription policies to all subscriptions to these publications, whether they are local or global.

Figure 4.2 depicts the logical structure of 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. More information about SMCPs can be found in Section 4.4.4. The fourth database is a module repository that contains the modules that are used to provide security for both the data plane and the security management plane.

As Figure 4.2 shows, a leaf-SMS has three interfaces. First of all it has a user interface where operators can issue commands. Secondly, it has an interface to its parent interior-SMS called a `Parent SMS Communicator`, a two way interface where commands and information can flow both ways.

Finally there is the main interface to the data plane called the `Data Plane Communicator`. Its task is to handle the two-way communication with the publishers and subscribers in the leaf-SMS' cloud. The inbound communication can be two things, either a policy the data plane node wants to get updated, or a request for a module that the originator needs. 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.

In the case of policy updates, the requests are passed up to the `Policy Engine` which performs the update and returns the updated policy through the `Transmuter` again, while in the case of the module request, the module is retrieved from the module repository if it exist there. If the module repository does not have it, the requests is sent to the `Parent SMS Communicator` which will return the module if it exist in any SMSs in the hierarchy.

The `Policy Engine` is the main logical component in the leaf-SMS. It controls 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`. If it gets a publication policy that does not yet exists, it retrieves the SGP for the security group that the publication policy specifies it wants to be

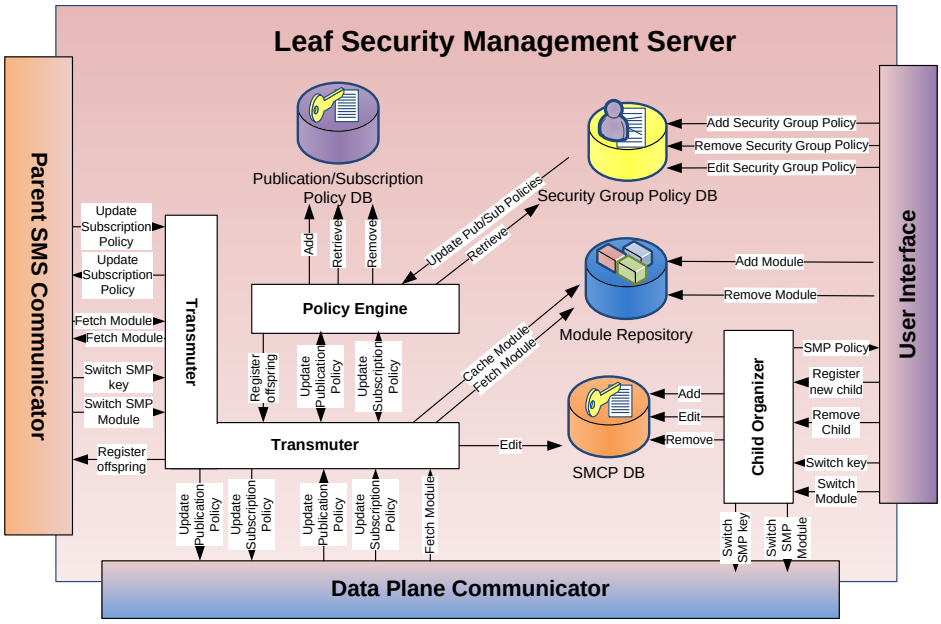

Figure 4.2: Leaf Security Management Server

a part of, and completes the policy with the information specified there, such as to set an expiration date on the policy and assign 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. More on the use of key generator modules can be found in Section 4.6.1.

In addition, to respond to data plane requests, the `Policy Engine` also handles changes done to the SGPs through the user-interface. Whenever a SGP is changed the `Policy Engine` propagates these changes to all publication and subscription policies in its policy database that are associated with that SGP. It can also propagate these updates down to the publishers and subscribers that are already using the affected policies, if that is desired by the operator. More on that can be found under *Forced* and *Natural* policy updates in Section 4.5.2 and 4.5.1 respectively.

Outgoing communication to the data plane consist of sending forced policy update commands

and applying SMCP changes for the different communication links with the respective children data plane nodes. SMCP changes are accomplished by switching keys and/or switching to another module according to the protocols defined in Section 3.3.

The `Parent SMS Communicator` controls all communication with the parent interior-SMS and is identical for any SMS, whether it is a leaf or interior. It can send and receive subscription policy updates and module download requests that cannot be handled locally, which enable subscription policies and modules to be routed through the hierarchy. It also has a `RegisterOffspring` command that makes it possible for the leaf-SMS to register the data plane nodes that it serves with its parent. In addition the `Parent SMS Communicator` can receive SMCP change protocol commands. These commands change the policy for the virtual point-to-point communication with the parent.

The operator controls the leaf-SMS through the user interface. Here he, or she, can define new security groups or edit existing ones, add and remove modules, and register child nodes with their associated SMCP.

4.1.3 Interior Security Management Server (interior-SMS)

A interior security management server's task can be compared to that of a DNS server. They are organized in a hierarchy and do not store any publication or subscription policies themselves, but exist to route subscription policies and modules on behalf of leaf-SMSs. This means that a interior-SMS does not need a `Policy Engine` or a `Publication/Subscription Policy Database` as the leaf-SMS. Neither does it have a `Data Plane Communicator`, but instead it has a `Child SMS Communicator` which enables it to communicate with its children SMSs.

Figure 4.3 depicts the logical structure of a interior-SMS. It has three databases, two of which it has in common with the leaf-SMS, namely the module repository and the SMCP-DB. The SMCP-DB is obviously needed to communicate securely with its children and parent, in the same way as the leaf-SMS talks to its parent interior-SMS and data plane nodes. The interior-SMS need

a module repository for two reasons: to store the modules assigned in the SMCPs and to cache modules that are being routed through the hierarchy.

The third database, called `Publisher to Child DB`, which is unique to the interior-SMS, stores the name of all the publishers registered in its scope. The leaf-SMS registers its publishers with its interior-SMS parent which stores the publisher associated with the child that serves it in the `Publisher to Child Database` and forwards the registration to its parent.

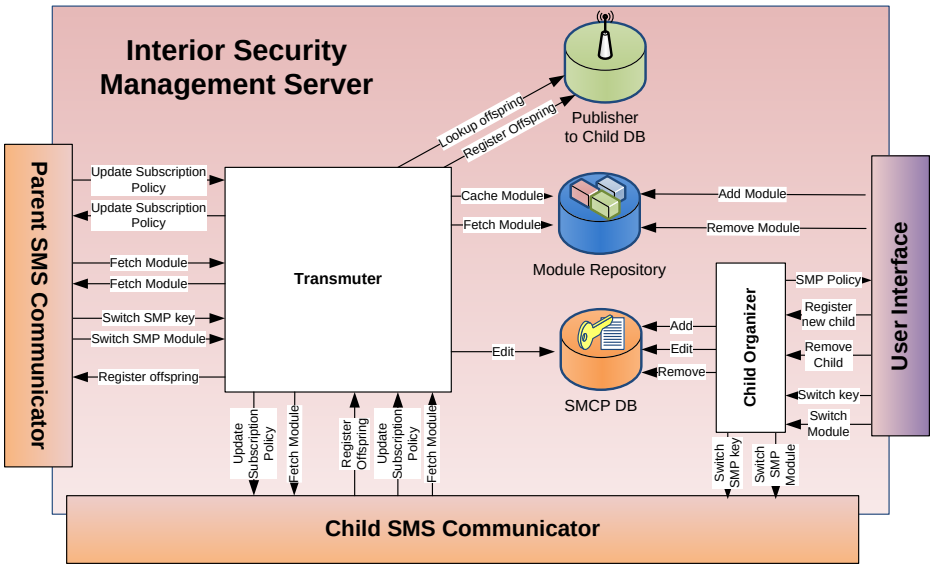

Figure 4.3: Interior Security Management Server

The user interface of a interior-SMS is a limited form of the user interface of a leaf-SMS. Since an interior-SMS does not control any data plane policies, the user interface only needs to support the adding and removing of modules and the management of the virtual point-to-point communication lines with its children SMSs. This is done through the same Child Organizer as the leaf-SMSs use to manage their communication with the data plane nodes.

4.2 Data Plane Extensions

To support the use of security modules and the publication and subscription policies when they are assigned, the GridStat's data plane had to be extended. Since the security architecture leverages an

end-to-end approach these extensions are limited to the publishers and subscribers that are the end points of the data plane status update streams.

4.2.1 Publisher

A publisher in GridStat is a producer of, with a few exceptions, 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.

To support the security architecture presented in this thesis the GridStat publisher component has been extended and made more modular. The design of the publisher has moved from the original all-in-one design depicted in Figure 4.4 to the extended version in Figure 4.5.

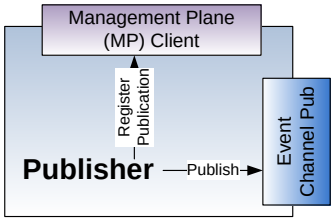

Figure 4.4: Original GridStat Publisher

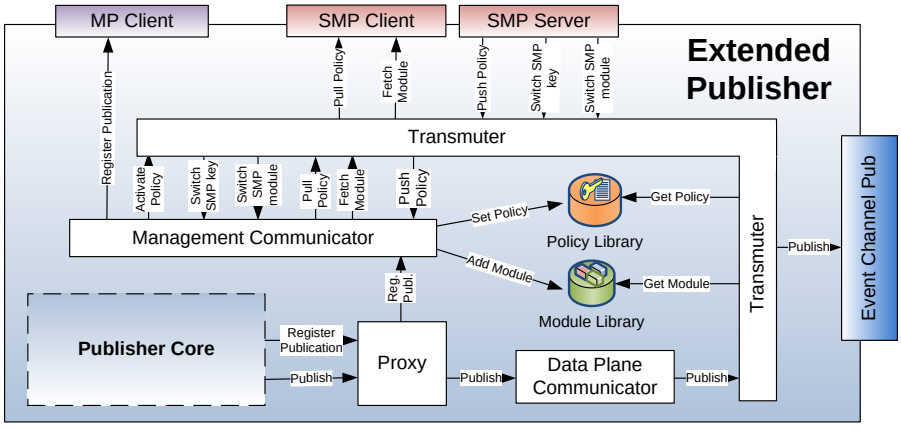

Figure 4.5: Extended Publisher

In the extended publisher the publisher core has the same commands available as the original publisher, but instead of directly contacting the management plane and pushing data into the data plane, all communication is sent to a proxy which delegates the tasks to different components. To set up new publications the core sends a register-publication command to its proxy who forwards the command to the `Management Communicator`. In order to successfully start publishing a new publication, the `Management Communicator` then has to register the new publication with the management plane, and get assigned a publication security policy by sending a *pull policy command* to the security management plane extension. All communication with the security management plane goes through the `Transmuter` which applies a security module according to the currently assigned SMCP to secure the communication between the publisher and its leaf-SMS. More details about the *Transmuter* and SMCPs can be found in Section 4.3 and 4.4.4 respectively.

When the `Management Communicator` receives the assigned publication policy, it checks whether the local `Module Library` has some or all of the modules that were assigned, and then sends a fetch module command to the security management plane for any that are missing. Finally the policy can be added to the `Policy Library`, activated in the `Transmuter` and the publisher core can start publishing through the proxy and into the now secured status update stream. More information about how policies are activated can be found under the policy propagation Section 4.5.

The two libraries keep the local dynamic security content organized. The `Module Library` stores all the downloaded modules, so they can be used whenever a policy assigns one of them without needing to download them again, while the `Policy Library` keep the records of the currently active publication policy for each of the publisher's publications and the SMCP needed to securely communicate with its leaf-SMS.

4.2.2 Subscriber

Subscribers are consumers of, with a few exceptions, rate-based status updates. The most notable exception is the already mentioed RPC system Ratatoskr [35]. A subscriber requests a subscription, with QoS requirements, to a published variable from the management plane. If the management plane accepts the request, the subscriber will start receiving status updates with the specified QoS guarantees from the status router that functions as the subscriber's connection point with the data plane. A subscriber also has functionality to retrieve the currently available modes and handle mode changes that potentially change subscription properties [10].

In this project the GridStat subscriber has been extended and made more modular to support the new security architecture for the data plane. The design of the subscriber has been moved from the all-in-one architecture shown in Figure 4.6 to the extended subscriber depicted in Figure 4.7.

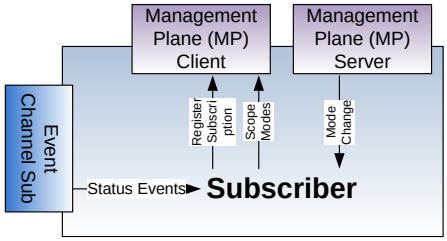

Figure 4.6: Original GridStat Subscriber

The extended subscriber design mirrors that of the extended publisher and reuses many of the same components. As with the extended publisher the extended subscriber lets the core keep all the functionality from the original subscriber, but lets it communicate with a proxy instead of directly with the external interfaces. The proxy delegates the commands to whatever component that should handle the command.

The process of registering a subscription is similar to the way a publication is registered with the exception of how the policy is activated in the `Transmuter`. The registration process is described in the previous publisher Section and more on the activation process can be found under

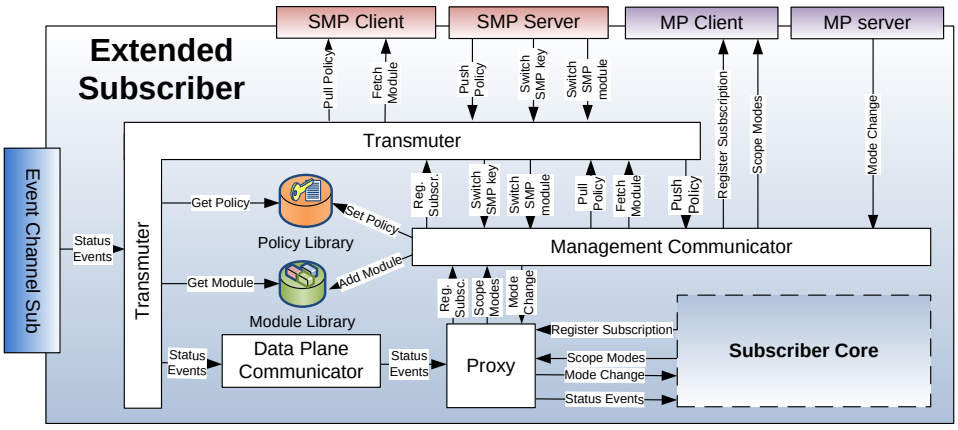

Figure 4.7: Extended Subscriber

the policy propagation Section 4.5.

4.3 Transmuter

Transmuters are objects embedded in all publishers, subscribers and SMSs that enforce the policies for the communication links. They take the messages sent over these links and apply the modules and keys assigned to that communication link specified in the respective policy files on each end point. Figure 4.8 show the `Transmuter` simple `transmute` method responsible for handling data plane traffic.

The `transmute` method performs three main tasks. First, it provides subscribers with the ability to perform reactive module switches described in Section 4.5.3. Secondly, it performs its main task of applying all the modules and keys assigned to the publication of which the status variable is a part and returns the transmuted result.

Finally the method checks whether the policy currently in effect is expired and needs an update. Since downloading a new policy can take a long time, and the `Transmuter` needs to be ready to handle more updates, it does not perform the update itself, but initiates a new thread to perform the task as described in detail in Section 4.5.1.

```
public ByteBuffer[] transmute(ByteBuffer[] bufferArray,IntHolder length)
{
        ByteBuffer[] ret = bufferArray;

        //Extracting the variable id from the status update
        int variableId = bufferArray[0].getInt(Constants.UPDATE_VARIABLE_ID_OFFSET);
        //Extracting the policy stamp from the currently used policy for this variable
        byte policyNum = m_pl.getPolicyNum(variableId);

        if(m_type==PUBLISHER)
        {
                //Stamping packet with policy number
                bufferArray[0].put(Constants.UPDATE_POLICY_STAMP_OFFSET,policyNum);
        }
        else
        {
                //Extracting the policy stamp from received update
                byte policyStamp = bufferArray[0].get(Constants.UPDATE_POLICY_STAMP_OFFSET);

                //Checking whether policy stamp corresponds to the current policy
                if(policyNum!=policyStamp)
                {
                        //Forcing a switch to the new policy
                        if(!m_pl.activatePolicy(variableId,policyStamp))
                                return bufferArray;
                }
        }

        //Retrieving the assigned modules
        Module[] modules = m_pl.getModules(variableId);

        //Iterate through the modules and apply them to the update
        for(int i=0;i<modules.length;i++)
        {
                try{ret = modules[i].performAction(ret);}
                catch (ActionFailedException e)
                {
                        System.out.println("WARNING: Unable to transmute ByteBuffer: "+e);
                }
        }


        //Check to see if the policy is expired and need to be updated
        if(m_pl.policyExpired(variableId))
        {
                PolicyReplacerThread prt = new PolicyReplacerThread(
                                                variable
                                                PolicyConstants.MODULE_SWITCH_DELAY);
                prt.start();
        }

        return ret;
}
```

Figure 4.8: The Transmuters transmute method

4.4 XML policies

The security architecture uses XML policies in order to provide a unified way of representing dynamic security aspects. These policies can easily be transferred between components can also be stored in persistent databases that allow for backup state replication.

Five different types of XML policies have been developed; the publication policy, subscription policy, security group policy (SGP), security management communication policy (SMCP), and module policy. While the first four specify assignment of security rules, such as modules and keys,

for some given type of communication lines, the module policy differentiates itself somewhat by describing security module attributes.

4.4.1 Publication Policy

Publication policies are XML structures that specify a set of security rules for a publication. Figure 4.9 shows the organization of this XML structure. It consist of 9 main nodes; type, time frame, policy number, publication name, publication id, publisher name, security group, assigned modules and a change log. This simple structure makes it easy to extend and add more functionality in future iterations and development. One example of such an extension, an ongoing project using TrustBuilder [38] to achieve dynamic user level access control, is explored in Section 6.2.7.

```
<GridStatPolicy>
    <Type>Publication</Type>
    <ValidTimeFrame>
        <From />
        <To />
    </ValidTimeFrame>
    <Modules />
    <PolicyNum />
    <Publication />
    <PublicationId />
    <Publisher />
    <SecurityGroup />
    <ChangeLog />
</GridStatPolicy>
```

Figure 4.9: The Publication Policy XML Structure

Publication policies are initially created by a publisher that wants to publish a variable. The publisher specifies the name and the id of the variable, its own name as publisher, the name of the desired security group with which the publication should be associated. It sets the policy number to zero and adds a *Created* entry in the change log. An example of such a policy before it is sent the SMP is shown in Figure 4.10. The incomplete policy is then sent to the publisher's leaf-SMS.

The leaf-SMS then fills in the missing information based on the current policy specified for the relevant security group. First it assigns the modules and generates keys, then it sets the valid time frame for the policy by taking the current time in milliseconds and adding the expiration time specified in the security group policy. Finally it updates the policy number and adds a modification

entry in the change log. An example of how a publication policy might look after the assignments can be seen in Figure 4.11.

The policy number in the policy is a sequential 8 bit number that is updated each time the policy is changed. By letting the publisher stamp the status updates that it publishes with this policy number, the subscribers can check which version of the policy was applied to the update and thus be able to switch to the new versions of the policy at the correct times. More on this can be found in the policy propagation Section 4.5.

```
<GridStatPolicy>
    <Type>Publication</Type>
    <ValidTimeFrame>
        <From />
        <To />
    </ValidTimeFrame>
    <Modules />
    <PolicyNum>0</PolicyNum>
    <Publication>status1</Publication>
    <PublicationId>1587288427</PublicationId>
    <Publisher>c1.pub1</Publisher>
    <SecurityGroup>0</SecurityGroup >
    <ChangeLog>
    <ModificationEntry
            Time="1189029273264"
            Author="c1.pub1">
            <Modification>Created</Modification>
        </ModificationEntry>
    </ChangeLog>
</GridStatPolicy>
```

Figure 4.10: A Publication Policy after its creation by a Publisher

4.4.2 Subscription Policy

Subscription policies are the subscription side equivalent of the publication policy. They are XML structures that contain the security specifications for single subscriptions and thus have many similarities with publication policies. As Figure 4.12 shows, the structure is almost identical to that of the publication policy structure shown in Figure 4.9. The only differences are the type is *Subscription* and the added `Subscriber` node where the name of the subscriber can be stored.

Subscription policies are initially created by a potential subscriber that wants to subscribe to publication. The subscriber adds as much information as it has access to when it creates the policy, before sending it to its leaf-SMS. Figure 4.13 is an example of how such a policy might look before it is sent to the SMP.

```
<GridStatPolicy>
    <Type>Publication</Type>
    <ValidTimeFrame>
        <From>1189115183367</From>
        <To>1207259183367</To>
    </ValidTimeFrame>
    <PolicyNum>1</PolicyNum>
    <Publication>status1</Publication>
    <PublicationId>1587288427</PublicationId>
    <Publisher>c1.pub1</Publisher>
    <SecurityGroup>0</SecurityGroup>
    <Modules>
        <Module Type="Obfuscation">
            <Name>AesObfuscation</Name>
            <Key>N/A</Key>
        </Module>
        <Module Type="Encrypt">
            <Name>Blowfish</Name>
             <Key>
               144934833588806222429984722844170914795649793255 1
             </Key>
        </Module>
    </Modules>
    <ChangeLog>
        <ModificationEntry
                Time="1189115183335"
                Author="c1.pub1">
                    <Modification>Created</Modification>
        </ModificationEntry>
        <ModificationEntry
                    Time="1189115183367"
                    Author="c1.sms">
                    <Modification>Assigned new module(s)</Modification>
                <Modification>Assigned new keys</Modification>
                <Modification>Modified the expiration time</Modification>
        </ModificationEntry>
    </ChangeLog>
</GridStatPolicy>
```

Figure 4.11: A complete Publication Policy

```
<GridStatPolicy>
    <Type>Subscription</Type>
    <ValidTimeFrame>
        <From />
        <To />
    </ValidTimeFrame>
    <Modules />
    <PolicyNum />
    <Publication />
    <PublicationId />
    <Publisher />
    <Subscriber />
    <ChangeLog />
</GridStatPolicy>
```

Figure 4.12: The Subscription Policy XML Structure

It is when the subscription policy reaches the leaf-SMS of the subscriber that the differences between publication and subscription policies start to show. Publication policies are all local to a single cloud and its leaf-SMS. A publication policy is therefore never sent outside the originating cloud, since only the publisher and its leaf-SMS need a copy of it, while a subscription policy needs to be routed to whatever leaf-SMS has the corresponding publication policy ,as described in Section 4.1.

Regardless of whatever routing is done to the subscription policy in the management plane, the policy will be completed and returned to the originating subscriber if the subscriber has the needed credentials to access the publication. As Figure 4.14 exemplifies, the completed policy contains the module assignments and the keys needed to decode the updates published by the publisher. Note that a subscription policy has the assigned modules in reverse order of the corresponding publication policy to be able to reverse the transmutation caused by the publisher's `Transmuter` applying its publication policy.

```
<GridStatPolicy>
    <Type>Subscription</Type>
    <ValidTimeFrame>
        <From />
        <To />
    </ValidTimeFrame>
    <Modules />
    <PolicyNum />
    <Publication>status1</Publication>
    <PublicationId>1587288427</PublicationId>
    <Publisher>c1.pub1</Publisher>
    <Subscriber>c1.sub2</ Subscriber>
    <SecurityGroup />
    <ChangeLog>
        <ModificationEntry
            Time="1189029270398"
            Author="c1.sub2">
            <Modification>Created</Modification>
        </ModificationEntry>
    </ChangeLog>
</GridStatPolicy>
```

Figure 4.13: A Subscription Policy after its creation by a Subscriber

A subscription policy also has the same expiration date as the corresponding publication policy it is associated with, to enable the use of a natural update pull scheme to propagate changes to security group policies down the data plane without losing any updates. More on how policies are updated can be found under natural policy updates and forced policy updates in Section 4.5.

```
<GridStatPolicy>
    <Type> Subscription </Type>
    <ValidTimeFrame>
        <From>1189115183367</From>
        <To>1207259183367</To>
    </ValidTimeFrame>
    <PolicyNum>1</PolicyNum>
    <Publication>status1</Publication>
    <PublicationId>1587288427</PublicationId>
    <Publisher>c1.pub1</Publisher>
    <Subscriber>c1.sub2</ Subscriber>
    <SecurityGroup>0</SecurityGroup>
    <Modules>
        <Module Type="Encrypt">
            <Name>Blowfish</Name>
            <Key>144934833588806222429984722844170914795649793255 1</Key>
        </Module>
        <Module Type="Obfuscation">
            <Name>AesObfuscation</Name>
            <Key>N/A</Key>
        </Module>
    </Modules>
    <ChangeLog>
        <ModificationEntry
            Time="1189029270398"
            Author="c1. sub2">
        <Modification>Created</Modification>
        </ModificationEntry>
        <ModificationEntry
            Time="1189029271249"
            Author="c1.sms">
            <Modification>Assigned new module(s)</Modification>
            <Modification>Assigned new keys</Modification>
            <Modification>Modified the expiration time</Modification>
        </ModificationEntry>
    </ChangeLog>
</GridStatPolicy>
```

Figure 4.14: A complete Subscription Policy

4.4.3 Security Group Policy (SGP)

Security groups abstract common security needs for groups of publications and thus make it easier to manage large groups of publication and subscription policies. Security groups can be created in the security management plane together with a SGP that specifies the security aspects of all the publications in that group.

Security group policies, or SGP's, are XML structures that specify the security aspects of security groups. These specifications are applied to all publications, and their subscriptions, belonging to the security group, by the leaf-SMS that originally defined that security group. As Figure 4.15 illustrates, the SGP is built up of only five nodes: policy type, modules, expiration length, and a change log. The `Modules` node holds the list and order of modules to be assigned to publications registered with that security group, the `Name` node holds the name of the security group, while the `ExpirationLength` specifies how often publications and subscriptions belonging to this group

needs to update their policy. Figure 4.16 exemplifies a completed SGP.

```
<GridStatPolicy>
      <Type>SecurityGroup</Type>
      <Modules />
      <Name />
      <ExpirationLength />
      <ChangeLog />
</GridStatPolicy>
```

Figure 4.15: The Security Group Policy XML Structure

```
<GridStatPolicy>
    <Type>SecurityGroup</Type>
    <Modules>
        <Module ">AesObfuscation </Module>
        <Module ">RSA</Module>
        <Module ">TripleDes </Module>
        <Module ">SHA512ErrorCheck</Module>
    </Modules>
    <Name>0</Name>
    <ExpirationLength>964130816</ExpirationLength>
    <ChangeLog>
        <ModificationEntry
            Time="1189029270398"
            Author="c1. sms">
            <Modification>Created</Modification>
        </ModificationEntry>
    </ChangeLog>
</GridStatPolicy>
```

Figure 4.16: A filled out Security Group Policy

Each security group, and corresponding policy, is defined locally for a single leaf-SMS and the publications within the cloud which it serves. Utilities are not very comfortable with letting others control the security and access to their sensitive information. By letting the operators of the leaf-SMS define the security groups for their own cloud through the leaf-SMS user interface, the security control is kept with the utility that operates that server. This makes each utility able to define their own security policies for their own publications, and thus increase their comfort level with sharing information.

Figure 4.17 illustrates how each leaf-SMS has its own sets of security groups, all data plane nodes that have at least one policy member of leaf-SMS A's security groups are colored blue, while all data plane nodes that have at least one policy member of leaf-SMS B's security groups are colored green. All publishers served by the same leaf-SMS must register their publications as members of one of that leaf-SMS's defined security groups, while subscription policies becomes

members of the same security group as the publication it subscribes to. This means *Sub-A1* that subscribes to the local *Pub-A1* will be a member of whatever blue security group *Pub-A1* is a member of, while the global subscription that *Sub-A2* has with *Pub-B1* makes it part of the green security group set controlled by leaf-SMS B.

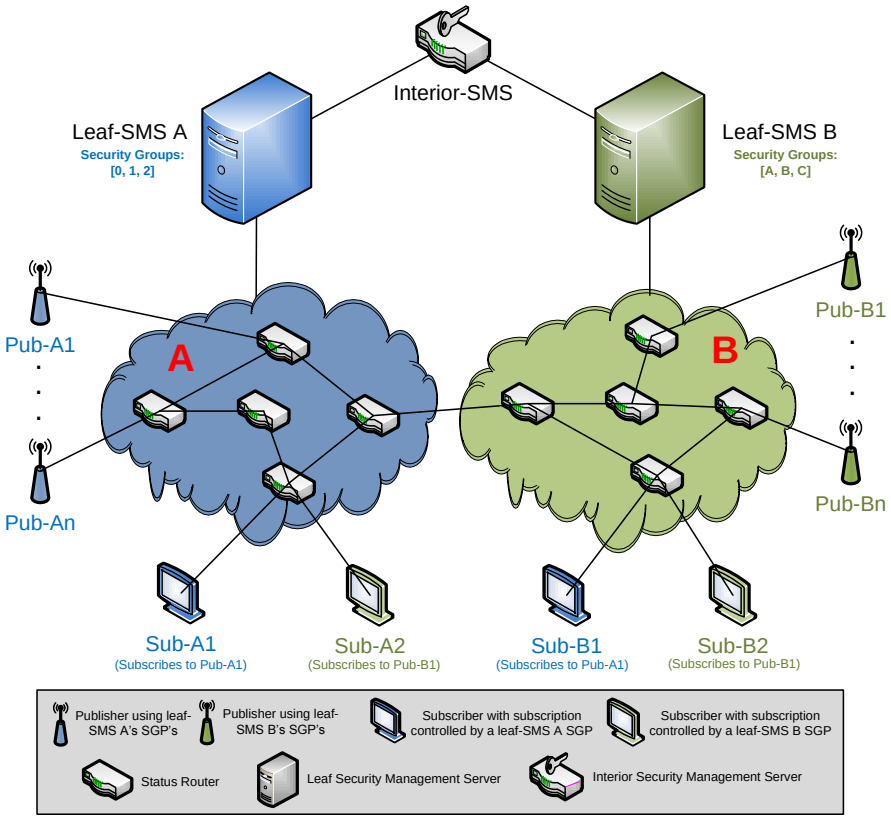

Figure 4.17: The local scope of SGPs

4.4.4 Security Management Communication Policy (SMCP)

To be able to provide dynamic data plane security, a new security management plane had to be put in place to manage it. As explored in Section 3.3 and 4.1 these virtual one-to-one management communication links also need to be secured in order to support the data plane security. The SMCP is a XML structure where these security aspects can be specified.

While Figure 4.3 presented the general types of virtual point-to-point links that SMCPs serve,

the design decision, explained in section 4.1.1, to organize the SMSs into a hierarchy and divide them into leaf-SMSs and interior-SMS, results in the slight type extension depicted in Figure 4.18. The general internal communication type is now replaced by communication links between a leaf-SMS and its parent interior-SMS, and a interior-SMS and its parent.

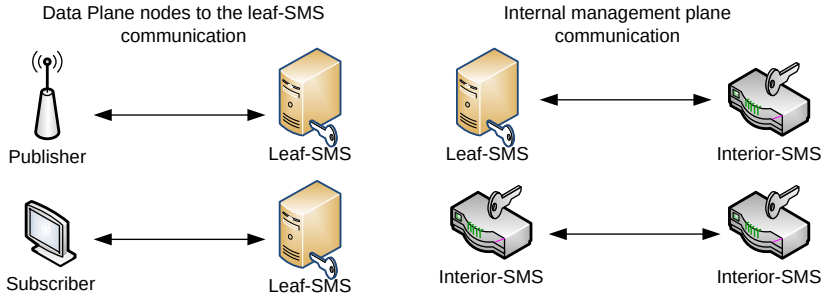

Figure 4.18: Types of Security Management Communication

Figure 4.19 depict the basic structure of the SMCP. The policy has a `Parent` node to specify the name of the parent, a `Child` node to store the name of the child, a `Modules` node to store the module assignments, and three nodes for the assigned keys. The `OutputKeys` node store the keys to be used for outgoing traffic, the `InputKeys` node for incoming communication and finally a `KeyIndex` node that holds the index of the currently used keys. By dividing the keys into output and input keys the policy is able to support the use of different keys for child-to-parent and parent-to-child communication, if that is desired.

```
<GridStatPolicy>
    <Type>SMCP</Type>
    <Parent />
    <Child />
    <Modules />
    <OutputKeys />
    <InputKeys />
    <KeyIndex />
</GridStatPolicy>
```

Figure 4.19: The Security Management Communication Policy XML Structure

SMCPs are defined by the operator that registers a new child with an SMS. A copy of the policy, together with copies of the specified modules, are initially provided out-of-band to the child node.

Later changes to the policy, such as key and module changes, are done in-band by taking advantage of the list of preloaded keys according to the protocols defined in Section 3.3.

An example of how a SMCP for the communication between a publisher and its leaf-SMS with 5 preloaded keys and using the AES security module is shown in Figure 4.20.

```
<GridStatPolicy>
    <Type>SmpCom</Type>
    <Parent>c1.sms</Parent >
    <Child>c1.pub1</Child >
    <Modules>
        <Module>
            <Name>AES</Name>
        </Modules>
    </Modules >
    <OutputKeys>
        <Key>112086096989480331912897997311082881488379932624 5</Key>
        <Key>573538035040423767260994354675225432755248146933</Key>
        <Key>116240351826497387206904501320861965003765879670 5</Key>
        <Key>118042816216492566325634072058730005942792119777 2</Key>
        <Key>766464578145571959657874755426490065157494018192</Key>
    </OutputKeys >
    <InputKeys>
        <Key>393865857916518092677659599425798904770321059830</Key>
        <Key>309494022550273857007681850407736755244636550858</Key>
        <Key>602044051055588754972889797158464857197272108 19</Key>
        <Key>140834833385320375300381871079348667927581712043 63</Key>
        <Key>250162622564474905520577543415412532866454770 48</Key>
    </InputKeys >
    <KeyIndex>0</KeyIndex >
</GridStatPolicy>
```

Figure 4.20: A Security Management Communication Policy example

4.4.5 Module Policies

Module Policies are different from the other policies by not specifying security rules for a single or group of communication streams, but instead specifying the properties of modules. When modules are added to the system through an SMS, the operator has to supply a series of attributes associated with the new module, which are stored in the module policy XML structure. The module policies specify the behavior of the modules and where the main copy is stored. As Figure 4.21 indicate, the module policy has 9 nodes defining attributes, such as the name of the module, the module type, its author, the date it was added to the repository, the size of its key, where the actual module file is stored, a description of the module, and the name of the key generator which should be used.

Figure 4.22 and figure 4.23 exemplify two different module policies. Figure 4.22 is an example of an encryption module that uses a 128 bit key generated by the default key generator, while Figure

```
<GridStatPolicy>
    <Type>Module</Type>
    <Name />
    <ModuleType />
    <Author />
    <DateAdded />
    <KeySizeInBits />
    <ModulePath />
    <Description />
    <KeyGenerator />
</GridStatPolicy>
```

Figure 4.21: The Module Policy XML Structure

4.23 shows an authentication module based upon RSA that uses a 2048 bit key and a standalone key generator module. The purpose of standalone key generator is defined in Section 4.6.1.

```
<GridStatPolicy>
    <Type>Module</Type>
    <Name>AES</Name >
    <ModuleType>Encrypt</ModuleType >
    <Author>Erik Solum</Author >
    <DateAdded>19-Aug-2007 12:30:23 PM</DateAdded >
    <KeySizeInBits>128</KeySizeInBits >
    <ModulePath>./ModuleRepository/AES</ModulePath >
    <Description>A 128 bit AES encryption module based
                 upon Suns standard JCE
    </Description>
    <KeyGenerator><<DEFAULT>></KeyGenerator >
</GridStatPolicy>
```

Figure 4.22: An AES Module Policy Example

```
<GridStatPolicy>
    <Type>Module</Type>
    <Name>RSA</Name >
    <ModuleType>Authentication</ModuleType >
    <Author>Erik Solum</Author >
    <DateAdded>19-Aug-2007 12:36:08 PM</DateAdded >
    <KeySizeInBits>2048</KeySizeInBits >
    <ModulePath>./ModuleRepository/RSA</ModulePath >
    <Description>A 2048 bit RSA encryption module based
                 upon Suns standard JCE
    </Description>
    <KeyGenerator>RSAKeyGenerator.jar</KeyGenerator >
</GridStatPolicy>
```

Figure 4.23: A RSA Module Policy Example

4.5 Policy Change Propagation

Changes done to SGPs in the security management plane have to be propagated down to the individual publication and subscription policies and be activated. This section presents the two

mechanisms that accomplish this: a natural pull mechanism, that with the use of expiration times provides an eventual propagation, and a forced push mechanism initiated from the security management plane that provides an immediate propagation. Both mechanisms employ the *reactive policy activation* approach described in Section 4.5.3 to minimize the chance of losing status update information during a policy switch.

4.5.1 Natural Policy Change Propagation

Natural policy updates are a pull model for propagating changes done to publication and subscription policies based on the fact that all policies have an expiration time, and that whenever a policy expires, the publisher and subscriber contact the security management plane for an updated policy. Any changes done to the SGP of the security group to which the publication belongs, will thus be propagated down to the publisher and subscribers of that publication.

Every security group policy has a set *max validation time* that is applied to all policies in that security group. Every publication policy that is created, or updated, is given an expiration time equal to the current time plus the max SGP validation time. All subscription policies associated with that publication policy will inherit this expiration time regardless of when the subscription was set up relative to the publication. This is done so the publication and all its subscribers will synchronize their policy updates. In order to support this synchronized clocks are assumed. GridStat already has functionality that relies synchronized clocks, so this is a acceptable assumption.

It is the `Transmuter` in each publisher and subscriber that initiates the natural policy updates by checking whether the policy is expired each time it has performed a transmutation with the policy. The actual update is done by a background thread so as to not interrupt the flow of messages as defined in the reactive policy activation Section 4.5.3.

When a publication policy is updated the expiration time is reset to the time of the update plus the max SGP validation time while the subscriber policies still are synchronized directly with current publication policy expiration.

4.5.2 Forced Policy Change Propagation

In some cases policy updates are so critical that it is not good enough to wait for the publishers and subscribers to update themselves naturally. While natural policy updates rely on predetermined times to update the policies, *forced* policy update forces the publishers and subscribers to switch to the new policy immediately. This is push approach in contrast to the natural updates pull approach.

A forced policy update can be initiated by operators that want to force the changes done to a SGP in a leaf-SMS to be propagated immediately. The `Policy Engine` will then start sending out the new updated policies of that security group to every publisher and subscriber that is affected by the new changes. An evaluation of the performance of this operation can be found in Section 5.3

The publishers and subscribers handle the forced policy update in the same way as the natural policy updates. Both will immediately start a background thread to download and initiate the modules assigned in the new policy. The publisher will activate the new module after a constant delay, to allow the subscribers to get ready, while the subscribers wait to activate the new modules until they receives the first update transmuted with the new policy.

4.5.3 Reactive Policy Activation

When switching from one set of modules to another in the data plane, it is important that both publishers and the subscribers synchronize their switch to avoid a temporary policy mismatch that would cause information loss. As explored in the previous two subsections, policy propagation can be initiated either by a push or pull approach, but are regardless handled by a background thread that works concurrently with the normal operation of the status update stream.

In the pull approach the background thread sends the currently used policy to the security management plane and gets returned an updated version, while in the push approach the updated policy is pushed down to the publishers and subscribers. Regardless of how the updated policy is obtained, the thread examines the updated policy and downloads any missing modules before

initializing them with the updated keys. When all the modules are ready the thread marks itself ready and goes to sleep before actually replacing the modules that are currently used. If the thread was started by a publisher, the thread will wake up to perform the switch on a given short delay after the expiration time. This is done to let the subscribers have time to get ready.

In the subscribers the switch is done based upon the policy stamp on the received updates. Each policy has a policy number that is incremented each time it is updated. The publisher stamps this number on all published status updates. The subscribers check this stamp and activate the waiting modules when they receive the first status update stamped with the new policy number.

The limitation to this solution is the delay between the expiration date and when the publisher makes the switch. If it is too long the operation takes unnecessarily long, but if it is too short and the subscribers are not ready to switch information is lost until the subscriber completes its switch. Work on the actual deployment of GridStat needs to be done to assert the optimal length of this delay constant.

4.6 Modules

In the implementation of the modular security architecture the modules are implemented as JAR files that contain all the class definitions needed to enable the Java Virtual Machine to load and instantiate an object that implements a known module interface in runtime with the help of an extension to its `ClassLoader` class. These modules can thus be distributed to the different nodes and loaded when they are needed, without significant interruption to the primary functions. As previously stated, the modules are primarily used as transparent interchangeable black boxes that provide different security mechanisms to both the data plane and the security management plane.

Modules can be added to the system through the user interface of any SMS and be automatically propagated based upon the need of the module. When a module is added the operator is required to specify a series of attributes for the module, which are stored in a module policy that becomes

associated with the module as specified in Section 4.4.5.

All JARs that are designed to be used as a modules have to contain a class that matches the name of the JAR file and that implements the predefined module interface presented in Figure 4.24. The interface forces all module classes to implement three `performAction` methods which provide the main interface between the different running GridStat components and the black box modules. Through these three methods the transmuters can apply whatever functionality the module provides to the bytes that are passed into it, either in the form of a status update wrapped in a `ByteBuffer` array or other types of information in simple byte arrays, or byte array holders. All three methods throw the `ActionFailedException` if the action the module is supposed to perform fails, e.g. if the module is suppose to decrypt, but finds the wrong number of input bytes.

```
public ByteBuffer[] transmute(ByteBuffer[] bufferArray,IntHolder length)
{
    public interface Module
    {
        public static final short INIT_SUCCESSFUL = 0;
        public static final short INIT_UNSUCCESSFUL = -1;
        public static final short DIRECTION_IN = -1;
        public static final short DIRECTION_OUT = 1;
        public static final short DIRECTION_INOUT = 0;

        public ByteBuffer[] performAction(ByteBuffer[] bba) throws ActionFailedException;
        public void performAction(OctetBufferHolder obh) throws ActionFailedException;
        public byte[] performAction(byte[] b) throws ActionFailedException;

        public short init(String initString,short direction);
    }
}
```

Figure 4.24: The Security Module Interface

The `init` method is used to initialize the module object with the needed input information, such as keys and whether the module object will be used on outgoing or incoming information. The information needed to do this is provided by whatever policy that specified the use of the module, and is usually in the form of a large number to be used as key. Note that not all types of modules need a *init* string, but all need to be initialized by calling the `init` method.

The following sections explore the different types of modules defined as relevant to achieving the goals of the security architecture. Section 4.6.1 takes a closer look on key generator modules while the rest focus on the five basic security module types defined in 3.2 and their proof-of-concept

implementations.

4.6.1 Key Generator

While most modules can use the standard hard-coded key generator, some modules require their keys to be generated in a specific way. This is supported by giving the operator the opportunity to specify a special key generator in the form of a second key JAR file. This JAR file contain the class definitions needed to instantiate a key generator class implementing the well-known *key generator interface*, shown in Figure 4.25, when he, or she, adds a new module to the system. This causes the specified key generator to be added to the SMS's module repository and a reference to it stored in the module's policy file.

```
public ByteBuffer[] transmute(ByteBuffer[] bufferArray,IntHolder length)
{
    public interface KeyGenerator
    {
        public String[] generateKeyPair(int keySize) throws NoSuchAlgorithmException;
    }
}
```

Figure 4.25: The Key Generator Interface

The Key Generator interface forces all key generators to implement a simple *generateKeyPair* method which takes the size of the key as input and returns the key as a string. This string can then be attached to a policy and later used to initialize an object of the module that was associated with it.

There can be different reasons to provide a special key generator. One such reason might be to provide a key generator that is not based upon the standard Java random generator, as the default generator is, but instead a generator that has a different and/or better source of pseudo random numbers. This further enhances the dynamic aspects of the security system since the dynamic modules do not need to rely on keys generated through a static key generator that might have weaknesses that are discovered after deployment.

Secondly some types of algorithms such as RSA need special forms of keys with more properties than being a random sequence of bytes. Since RSA is asymmetric it needs the keys used at

the end points to have a special relationship that just random sequences of bytes cannot provide. A specialized RSA key generator has been implemented and tested.

4.6.2 Encryption Modules

Encryption modules are the main type of modules used to secure GridStat's data plane. These are modules that encrypt data in one end of the streams, and decrypt in the other, to provide the stream with confidentiality.

Even though any encryption algorithm could be implemented as an encryption module the properties of GridStat poses some limitations. The use of multicast excludes the use of algorithms that differentiate the published status updates based upon the subscribers; the differences in rates between publishers and subscribers, imposed by the status router filtering, invalidates all algorithms that requires the producer and consumer to be synchronized, such as stream ciphers; and the generally strict latency demands further restrict what algorithms that can be used.

Based upon these limitations the use on symmetric block ciphers stands out as the best alternative being relatively fast, not needing synchronization and keeping the updates identical regardless of the subscribers. Five block cipher modules have been developed for demonstration purposes: a Caesar cipher module built from scratch to function as a benchmark module; and a DES, AES, Blowfish and Triple DES modules built on Sun's Java Cryptography Extensions (JCE) [37, 8]. These modules are in no way optimal, since they employ costly logic to translate beween the JCE and GridStat formats, and are only developed as a proof of concept to show that the real-time requirements of GridStat can be met using this modular approach.

4.6.3 Obfuscation Modules

The problem of keeping small data samples confidential is greatly increased if the content of the data samples are uniform and/or repeated. This enables an attacker to much more easily extract the encryption key [20]. Obfuscation modules try to mask these patterns by introducing randomness to the information. They do not encrypt the data, just mask some of the properties of the data when

combined with an encryption module, thus complementing the encryption module, not replacing it.

There are two main problems, repetition of the same data sample and uniform data samples. Repetition can be caused by a sensor measuring something that has slow rate of change, while uniform data can be a result of something as simple as a sensor using 0 as a default value when something is wrong.

There are many ways to obscure, or mask, these facts by introducing randomization. One way is to make the same data sample look different each time it is sampled by shuffling the content around based on random values. Since we are using random values we need to let the receiver of the sample, in GridStat's case the subscriber, know how it can reorder the data. The simplest way to do this is to embed the information needed into the original data. This can be allowed since the objective of the module is not encryption, only obfuscation.

It is possible to shuffle with any granularity. The finer the grain used, the more information is need to add to enable the subscriber to make sense of the information, but the better the obfuscation. Combined with the modular approach this allows the granularity to be tuned to the needs of each individual published variable. Figure 4.26 exemplifies a simplistic way that an 8 byte update could be shuffled using bytes as granularity.

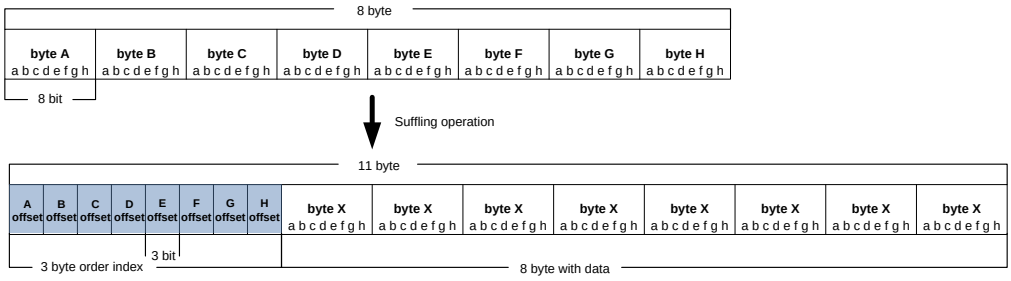

Figure 4.26: Obfuscation by shuffling

The shuffling approach has the weakness of its efficiency being a function of the uniformity of the data. In a worst-case situation with completely uniform data, such as only bits set to 0, shuffling

will have no effect. Other approaches are required to introduce randomness to the individual bits regardless of the uniformity of the original data.

One way to achieve real randomization is to perform standard block cipher algorithm operations with random keys that then are embedded with the update. Using block ciphers with random keys provides randomization between updates, and thus handles the problem of repeating data samples. Unfortunately block cipher's inherent problem that identical blocks always encrypt to the same block using the same key, still is a weakness. Since the key are replaced between each update this won't affect the masking of repeating data samples and uniform data across updates, but it does reduce the modules' ability to mask uniform data internally in each update, which opens up a chance of pattern attacks on individual updates.

An alternative to block ciphers is to introduce randomness by generating one-time-pads that get applied to the data through a simple algorithm such as the Caesar-Cipher, and then embedded in the update. Figure 4.27a presents an example of this approach being applied to an 8 byte data sample. Each byte is incremented with a byte from the one time pad, which then is embedded into the update. To avoid that the appended obfuscated byte becomes identical to the one-time-pad when the original byte is uniformly zeros, as byte 8 in Figure 4.27a exemplifies, a simple constant offset can be added as shown in Figure 4.27b.

One time pads are a special type of process-reversible obfuscation called *random number reversible* obfuscation[13]. This means that the only way the original data can be reverse-engineered from the obfuscated data, is to either obtain of the one-time-pad, or crack the random generator. Assuming the encryption module keeps the random one-time-pad confidential, the attacker thus need to crack the random generator used so he, or she, can gain knowledge of the next random number. The current implementation is based up the standard Java random generator and thus shares its strengths and weaknesses.

Two obfuscation modules are implemented and evaluated: one module based upon the block cipher approach using the AES algorithm called `AESObfuscation`, and one module using the

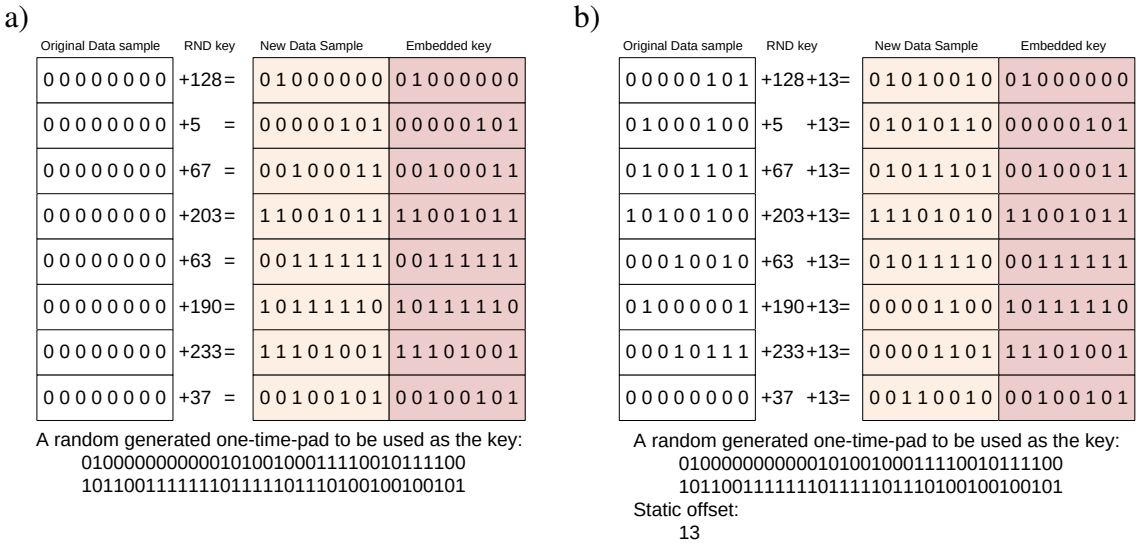

Figure 4.27: One Time Pad Obfuscation example, a) without offset and b) with an offset

one-time-pad approach called `OneTimePadObf`. Both are evaluated in Section 5.1.

4.6.4 Authentication Modules

While the encryption modules and obfuscation modules goals are to achieve confidentiality for the published variables from non-authorized subscribers, the authentication modules objective is to secure against attackers injecting false information into the status update streams. Authentication modules address this by letting the publisher sign the status updates with a digital signature that a corresponding module in the subscriber can verify.

One way to accomplish this is to use asymmetrical algorithms that uses a private key to sign the status update in the publisher, and a public key to verify it in the subscribers. Given a key of sufficient length, this ensures that the subscribers only accept messages signed by that private key. The problem with asymmetrical algorithms is that they need relatively long keys to be secure, and this makes the signing and verification operation relatively slow. As the evaluation of the implemented `RSA` module using a 2048 bit key will show in Section 5.1, its performance is not compatible with real-time delivery of information.

An alternative to using resource expensive asymmetric algorithms is to simply add a set of secret identification bytes to each update in order to sign them. Though such an approach in itself would achieve little, since an attacker easily could replicate such a signature, combined with encryption and obfuscation modules it can achieve much of the same effect that using asymmetrical algorithms can provide, but with the fraction of the cost.

Adding a secret signature to each update, and then applying an obfuscation module that randomizes the signature before it is encrypted by a third module, would make it extremely difficult for an attacker to inject falsified information. First of all it would need to know the secret signature and secondly, it would need to know the key used by the encryption.

A limitation inherent to a static signature approach is that, since it is based upon both the publisher and subscribers knowing the signature, it cannot secure against malicious subscribers. A subscriber could use their knowledge of the secret key to inject falsified data into the other subscribers. The `SimpleAuth` can only authenticate that the status update was sent from a node part of the key group. However as the implemented `SimpleAuth` show in Section 5.1, this is a tradeoff that is greatly outweighed by the much lower use of resources compared to using asymmetrical algorithms.

4.6.5 Integrity Modules

Error checking and error correcting modules aim to enhance integrity and availability. They do this by adding redundant information at the publisher, which is used to check/correct status updates at the subscriber. Integrity modules are usually not in need of a key as their work is not confidential.

The simplest forms of error checking modules are based upon hashing algorithms. Such modules run a hashing algorithm on the status updates in the publisher, and embed the hash value with the updates. The corresponding module on the subscriber side, extracts the hash value and repeats the hashing to check whether it matches. This provides an integrity check to the process and four such modules has been implemented for proof-of-concept purposes; the `Crc` module based

on simple cyclic redundancy checks (CRC), the `MD5ErrorCheck`, `SHAErrorCheck`, and the `SHA512ErrorCheck` module. These are evaluated in Section 5.1.

4.6.6 Filtering Modules

Even though the status routers perform filtering, and do not forward unneeded status updates, some desired attributes could be gained by also letting the publisher filter its outgoing status updates. In addition to generally reducing the edge status routers workload, that otherwise have to do all the filtering, it would provide a first line of defense against denial-of-service attacks from misbehaving publishers.

If a publisher, either by malicious intent or by human/machine error, starts pumping out updates at a too high rate, it could overload the edge status router and thus result in denial of service to other publications using that edge status router. Figure 4.28 a) and b) shows how a filter module placed in the publisher that is controlled by the management plane could efficiently handle this type of attack by only letting through updates that one or more subscribers needs.

a) Misbehaving publisher without a filter module b) misbehaving publisher w/ a filter module

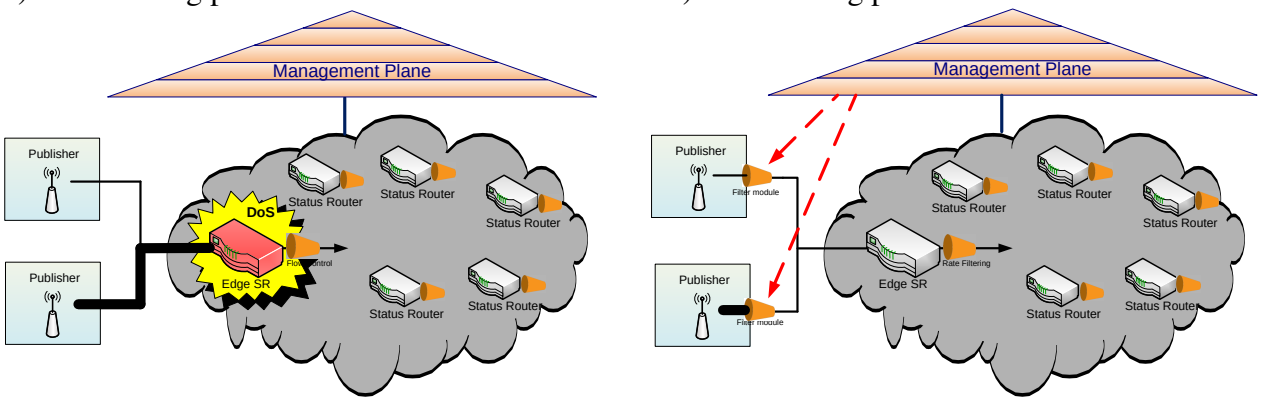

Figure 4.28: Filtering module

A filter module was developed and demonstrated for an early proof-of-concept version of the security system, but not carried on into the new version because of underlying conflicts with how the current status router rate filtering uses the timestamps in the data plane. Future work is needed

to work out a way for this conflict to be resolved and to avoid using timestamp manipulation work-arounds similar to what was used in the earlier proof-of-concept system.

CHAPTER FIVE

EVALUATION

This Chapter presents experiments that show that the modular security architecture presented in Chapter 3 and 4 not only are able to provide over-the-wire configurable security, but also adhere to the real-time requirements of the power grid and handle the scale needed. The following aspects are considered:

- The level of increased end-to-end latency on the status updates in the data plane caused by the security extension and the execution of the different proof-of-concept security modules.
- The resulting decreased data plane throughput and increased bandwidth usage.
- The scalability of publication and subscription policy propagation latency, both with respect to the volume of policies and the number of levels in the security management plane.
- Security management communication policy propagation latency.

The industry standard IEC 61850, also called *GOOSE*, define 4 milliseconds as maximum real-time latency between intelligent controllers [15]. While the 4 millisecond real-time requirement of the power grid impose strict requirements for the end-to-end latency in the data plane, the latency requirements for policy propagation are more relaxed and more focused on scalability. In an effort to address these two emphases the evaluation is divided into two sections: Section 5.1 evaluates the data plane performance with individual and combination of modules, and Section 5.3 evaluates the performance of the policy propagation infrastructure.

5.1 Data Plane Performance

For a security architecture to be a viable solution for an information system in the power grid it needs to be able to provide the necessary security as well as keeping the latency within acceptable levels. This Section will show that the end-to-end approach taken by the presented security

architecture allows combinations of the implemented proof-of-concept modules in table 5.1 to accomplish this in a modular way.

Each proof-of-concept module has different security and performance properties. They can thus be combined into sets that provide the best trade-off between different security aspects and performance for each published variable.

5.2 Data Plane Testbed

To run the data plane performance experiments a simple GridStat topology was used as shown in Figure 5.1: a single leaf-QoS Broker, one leaf-SMS, one status router, one publisher, and one subscriber. A single publication was set up between the publisher and the subscriber. The whole setup was run on a single machine with a 2.4 GHz Intel E6600 dual-core with 2 gigabytes of RAM running Ubuntu Linux with kernel 2.6.20-16. All GridStat components where compiled and run with java2SE 6 (version 1.6.0-02).

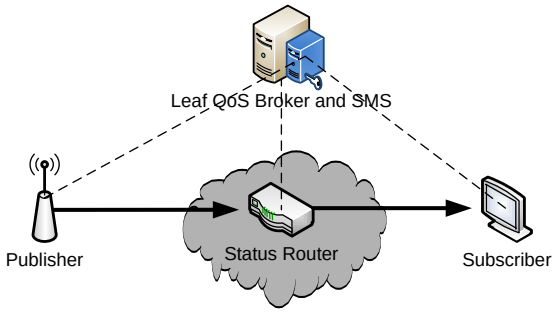

Figure 5.1: Data Plane Performance Test Setup

An inherent problem with measuring Java performance is its use of garbage collection to handle memory de-allocation. The garbage collector locks down the process when it executes and thus causes jitter in the measured performance. In an effort to mitigate this all the spikes caused the garbage collector are filtered out from the presented results. The unfiltered data can be found in appendix A.

Name	Type	Size[a]	Description
CaesarCipher.jar	Confidentiality	4598	A simplistic proof-of-concept module employing the Caesar cipher algorithm to achieve symbolic confidentiality
DES.jar[b]	Confidentiality	5427	A Data Encryption Standard[6] implementation with a 64 bit key of which 58 bit is effective
Blowfish.jar[b]	Confidentiality	5417	An implementation of the Blowfish algorithm[34] using a 128 bit key
AES.jar[b]	Confidentiality	5403	A module implementing the Advanced Encryption Standard [9] algorithm using a 128 bit key
TripleDES.jar[b]	Confidentiality	5418	Module implementing Advanced Encryption Standard algorithm [7] using a 160 bit key of which 128 bit is effective
AESObfusccation.jar[b]	Obfuscation	5447	Module that uses the AES encryption in combination with random keys that are embedded into the information
OneTimePadObf.jar	Obfuscation	4377	Module that uses random generated one time pads that are applied and then embedded into the data.
RSA.jar[b]	Authentication	5976	Module implementing the RSA algorithm[33] with a 2048 bit private key to sign data at the publishers and a public key to authenticate in the subscribers.
SimpleAuth.jar	Authentication	5976	Module that uses a generated static 64 bit signature to achieve increased performance over the use of asymmetrical algorithms.
Crc.jar	Integrity	5202	Module implementing the a 16 bit CRC algorithm.
MD5ErrorCheck.jar[b]	Integrity	5976	Module implementing the MD5 128 bits hash algorithm[b][31] to ensure the integrity of the data.
SHAErrorCheck.jar[b]	Integrity	5975	Module implementation of the SHA 160 bits hash algorithm[b][5] to ensure the integrity of the data.
SHA512ErrorCheck.jar[b]	Integrity	5992	Module implementing the SHA 512 bits hash algorithm[b][5] to ensure the integrity of the data.

Table 5.1: The proof of concept modules implemented and evaluated

[a]Module sizes are given in bytes

[b]Modules implemented based on the Suns JCE [37, 8]

In each experiment the system was allowed to send 30,000 status updates before the performance was measured. This is done to allow the Java virtual machine to optimize the execution of the code. Then the latency for the next 10,000 status updates was measured in the publisher and the subscriber.

5.2.1 Latency

By employing an end-to-end approach the latency associated with each module becomes constant relative to the length of the status update paths and the added latency they cause can thus be presented without taking the path length into account as shown in Table 5.2. Since the size of the status update obviously affects the number of processor cycles each module needs to perform their task, an experiment is run for each of the modules with a standard integer update and a 200 byte string. The status updates would in a deployment of GridStat represent readings from sensors and these readings are as a norm primary types such as integers, longs and floats, but in rare cases larger status updates would be needed and thus supported.

The proof-of-concept modules tested in this Section are non-optimized modules of which most are based on Sun's JCE, which has a very limited interface that is not compatible with GridStat's status update format. As a result the modules need to employ costly translation logic between the two formats that copies the data at least two times. This means that the numbers presented in Table 5.2 and 5.3 have a lot of potential for improvement, but even so, the security architecture is able to provide security within the real-time requirements.

As presented in Table 5.2 the baseline `CaesarCipher` module performs much better than the other encryption modules. This is partly due to the `CaesarCipher's` obvious simplistic algorithm, but also the fact that it is build from the ground up and does not need to copy the status update data to translate back and forth to another format. Of the nother encryption modules, `Blowfish` performed best with `AES` as a close second. The experiments show that both can

Module	Publisher side latency		Subscriber side latency		End-to-end latency	
	Integer	**200 bytes**	**Integer**	**200 bytes**	**Integer**	**200 bytes**
CaesarCipher	2,856	3,295	3,224	4,844	6,079	8,139
Blowfish	7,336	16,091	9,318	32,263	16,654	48,354
AES	8,341	16,203	8,101	33,231	16442	49,434
DES	8,059	22,945	9,561	42,486	17,620	65,431
TripleDES	9,748	43,642	12,473	70,439	22,221	114,081
OneTimePadObf	8,303	34,314	7,489	28,314	15,792	47,959
AESObfusccation	52,804	57,260	34,134	58,800	86,938	116,060
SimpleAuth	2,161	2,189	2,332	2,414	4,492	4,603
RSA	$114*10^6$	$114*10^6$	933,402	944,225	$114*10^6$	$114*10^6$
Crc	2,589	16,789	2,654	17,075	5,243	33,863
MD5ErrorCheck	5,197	7,721	4,815	6,804	10,002	14,525
SHAErrorCheck	6,490	9,823	5,938	9,423	12,428	19,246
SHA512ErrorCheck	11,051	17,698	12,481	19,028	23,532	36,726

Table 5.2: Proof-of-concept modules and the underlying module logic latencies in nanoseconds

apply a 128 bit encryption on the stream of integers while adding below 17 micro seconds of end-to-end latency and handle the 200 byte stream within 50 microseconds. `DES` and `TripleDES` on the other hand performed much poorer with the larger data samples. DES with only 58 bit effective encryption had an average of 65 microseconds latency on the 200 bytes status update and `TripleDES` used as much as 114 microseconds on its status updates. Clearly the `AES` and `Blowfish` modules are the better choice for achieving confidentiality.

The two evaluated obfuscation modules follow the two approaches presented in Section 4.6.3. The `AESObfuscation` module randomly generates a new key for each status update and uses this key to encrypt it with the AES algorithm before the random key is appended to the status update. As the experiments show this caries a great cost on performance. Since a new key is used for each status update the module need to re-initialize its encryption cipher for each event, the performance gets much worse than that of the `AES` encryption module that uses the same key each time. This corresponds to the findings done by Opyrchal and Prakash [28]. The `OneTimePadObf` module on the other hand performs much better, especially with small status updates, which is

natural since the one time pad approach doubles the size of the status update. The integer experiment shows that data in a status update can be completely obfuscated within only 16 microseconds and since the chance of repeating status updates are greater on smaller data samples one time pad obfuscation seems the more viable obfuscation alternative.

The experiments show that the two authentication modules, `RSA` and `SimpleAuth`, based on the asymmetric RSA algorithm and the static signature scheme presented in 4.6.4 respectively have hugely different performance impacts. The `RSA` module with a 2048 bit key adds as much as 113 milliseconds of latency which is in strict violation with the need for real-time delivery of information, while the `SimpleAuth` because of its simplistic logic only adds 4.5 microseconds in the 200 bytes experiment. This shows that it is unlikely that asymmetric authentication can be used in situations where there are real-time requirements, but that simpler forms of modules in combination with other modules such as obfuscation modules could be used to achieve close to the same level of authentication.

The four integrity modules implemented and evaluated are error checking modules, three of which that uses different hash algorithms to assert the integrity of reviewed information. The `MD5ErrorCheck` module generates a 128 bit hash, the `SHAErrorCheck` uses 160 bit hash and the `SHA512ErrorCheck` module employs a 512 bit hash of to achieve a varying degree of integrity checks. The varying length of the hash impacts the added end-to-end delay, but they scale well from the integer size experiments to the 200 bytes experiment and show that a 200 byte sized status update on average can be error checked with either a MD5 hash or a SHA hash within 20 micro seconds end-to-end. The `Crc` module behaves differently that the hash algorithms. It is very fast on small data sizes, but the performance quickly deteriorate when the data size is increased.

To test how the performance was affected by combining modules into sets the experiments in presented in Table 5.3 were undertaken. As the two first experiments show, the order in which modules are organized affects the latency they add. This is due to the different inflation of the data size, e.g. applying a hash module before applying an encryption module will require the

#	Module set	Publisher side latency		Subscriber side latency		End-to-end latency	
		Integer	**200 bytes**	**Integer**	**200 bytes**	**Integer**	**200 bytes**
1	MD5ErrorCheck Blowfish	16,612	27,208	14,736	35,231	31,348	62,439
2	Blowfish MD5ErrorCheck	12,746	24,588	10,412	34,434	23,158	59,022
3	OneTimePadObf Blowfish MD5ErrorCheck	14,042	54,404	14,736	52,885	28,778	107,289
4	OneTimePadObf AES SHA	15,376	60,701	15,946	60,111	31,322	120,813
5	SimpleAuth OneTimePadObf Blowfish	13,393	46,717	11,392	39,115	24,785	85,833
6	AESObfuscation TripleDES SHA512	70,602	135,048	59,304	108,073	129,906	243,121
8	SimpleAuth OneTimePadObf Crc Blowfish	15,765	121,353	13,280	96,782	28,348	218,136
7	SimpleAuth OneTimePadObf Blowfish SHA	15,068	59,583	16,577	49,986	32,342	109,569

Table 5.3: Module set latencies in nanoseconds

encryption module to encrypt the hash and thus increase the latency. Section 5.2.3 explores the modules different data inflation rates in more detail.

Experiment 3 and 4 in Table 5.3 show that combinations of one-time-pad obfuscation, Blowfish/AES encryption and MD5/SHA integrity modules can be applied to a stream of integers status updates with an average added latency of below 32 microseconds and 120 microseconds for the 200 byte stream. Experiment 5, on the other hand, show that if we instead of error checking are interested in authenticating without integrity checks, the stream the added latency can be reduced down to 85 microseconds for the 200 bytes stream.

To show a worst case scenario, experiment 6 combines the three slowest non-asymmetric modules; `AESObfuscation`, `TripleDES` and the `SHA512ErrorCheck`. The added latency gets as high as 130 microseconds for integers and 243 for the 200 bytes. This clearly not a good combination of modules, but shows that as long as the number of processing cycles are constant related to the length of the status update stream, even this combination can be tolerated.

Experiment 7 and 8 takes it one step further and show that we are able to provide authentication, obfuscation, encryption and integrity by combining the `SimpleAuth`, `OneTimePadObf`, `Blowfish` and the `SHAErrorCheck` or `Crc` module. With the use of the `Crc` module the the security only adds 28 milliseconds of end-to-end latency comparative to the 32 microseconds with the `SHAErrorCheck`. On the larger 200 byte stream the situation is reversed. The `Crc` module does not handle the increased size of the data and is easily outperformed by the set of modules that employs SHA that only adds 110 microseconds end-to-end latency. This clearly shows that even with the non-optimized proof-of-concept modules evaluated here the modular approach is able to provide the security needed for the status update streams while adhering to the stated real-time requirements. Future optimization will further reduce the added latency and enable even higher levels of encryption.

5.2.2 Throughput

The theoretical throughput can be calculated by dividing $10 * 10^8$ by the number of nanoseconds latency shown in shown in Table 5.2 and 5.3. The most interesting throughputs, that of the module combinations, are presented in table 5.4.

Employing security measures costs resources and that is illustrated by experiment 6 that shows that the using the `AESObfuscation`, `TripleDES` and `SHA512ErrorCheck` modules together the theoretical throughput of the publisher according to the experiments is reduced to just 7405 200 bytes status updates per second. The corresponding subscriber is reduced to only 9,253 status updates of equal size per second.

On the other hand experiment 7 and 8 shows that with the right combination of modules it is possible to provide obfuscation, encryption, authentication and integrity control while having a theoretical throughput of 66000 integer status updates per second, or 16000 200 bytes updates per second. That would translate to roughly 40 Mbit for the integers and 110 Mbit for the 200 byte updates. These numbers, being a function of the latency numbers, share the latency numbers immense potential for improvement when optimized modules are developed as discussed in Section 6.2.1.

Another property that can be observed from Table 5.4 is that as a norm the publisher needs slightly more resources to obfuscate and encrypt the data than the subscribers need to normalize and decrypt the data.

5.2.3 Bandwidth

In addition to increasing the latency most security also inflate the size of the data needed to be transferred and thus increase the bandwidth usage. The different proof-of-concept modules all have varying degrees of data inflation. For some, such as the integrity modules, this is a constant increase, while for others, such as the `OneTimePadObf` module, the increase is a function of the size of the original data.

#	Module set	Publisher Throughput		Subscriber Throughput	
		Integer	**200 bytes**	**Integer**	**200 bytes**
1	MD5ErrorCheck Blowfish	60,197	36,754	84,432	28,384
2	Blowfish MD5ErrorCheck	78,455	40,671	96,039	29,041
3	OneTimePadObf Blowfish MD5ErrorCheck	71,213	18,381	67,863	18,909
4	OneTimePadObf AES SHA	65,038	16,474	62,711	16636
5	SimpleAuth OneTimePadObf Blowfish	74.669	21405	87,778	25,565
6	AESObfuscation TripleDES SHA512	14,164	7,405	16,862	9,253
7	SimpleAuth OneTimePadObf Crc Blowfish	66,366	8,240	75,301	10,332
7	SimpleAuth OneTimePadObf Blowfish SHA	63,433	16,783	60,324	20,006

Table 5.4: Module combination throughput

Table 5.5 showa how the modules impact the size of the status updates during the experiments. All the encryption modules with the exception of the `CasesarCipher` module increase the size of the status update by rounding up to the nearest byte size that can be divided by 8. This inflates the status update by the x bytes satisfying the following equation: $(orgSize + x) mod\ 8 = 0$, $0 < x <= 8$. Notice that the `AES` module deviates from the others by having a minimum cipher text size of 16 bytes, which result in a 12 byte increase for the integer status update. Above the 16 byte minimum cipher text size the `AES` module behaves in the same way as the other encryption modules.

Comparing the two obfuscation modules, the `AESObfuscation` module has a constant inflation on the status update size, but a higher starting point, while the one-time-pad module always doubles the size of the update. As a result, on status updates below 10 bytes the `OneTimePadObf` module will produce smaller updates, but on larger updates the `AESObfuscation` obfuscation becomes the more conservative. A way to remedy the linear size increase to a degree could be to extend the `OneTimePadObf` module with simple techniques to reduce the size of the one-time-pad such as repeating smaller pads. This would reduce the level of obfuscation, but also greatly decrease the blow up of the update size and could be tuned to fit the needs of each individual publication.

As Table 5.5 illustrates the two authentication modules does not only differ greatly in their use of CPU cycles, but also in their use of bandwidth. The 2048 bits `RSA` module inherently increases the size of the data to blocks of 258 bytes. The result of this is that no matter how small the original data it is signing the minimal bandwidth use is 258 bytes. In comparison the `SimpleAuth` module adds only the 8 byte digital signature no matter the size of the data. Again we can observe that the `SimpleAuth` module combined with other modules perform better that the conventional `RSA` module for small status updates with real-time delivery requirements.

The three hash based integrity modules obviously inflate the size of the data with exactly the size of their hash. The more secure the hash, the larger the data becomes and this reveals yet another

Module	Data Inflation in bytes	
	Integer	*200 bytes*
CaesarCipher	0	0
DES	4	8
Blowfish	4	8
AES	12	8
TripleDES	4	8
AESObfusccation	28	24
OneTimePadObf	4	200
RSA	256	56
SimpleAuth	8	8
Crc	2	2
MD5ErrorCheck	16	16
SHAErrorCheck	20	20
SHA512ErrorCheck	64	64

Table 5.5: Modules bandwidth usage

place where assigning security modules on a per-publication granularity enables optimization. The standard hash algorithms have been developed for large amounts of data, not small 4 byte status updates. This is illustrated by the `SHA` module in Table 5.5 which use a 8 byte hash to assure the integrity of a 4 byte status update. This is obviously not an optimized solution and the more simple cyclic redundancy check (CRC) module is more efficient, bot in respect to latency and bandwidth.

As the bandwidth experiments have shown the use of security modules comes with a cost on the bandwidth usage, but is within acceptable limits. The physical need for bandwidth is of course a multiplication of the bits of each update and the rate of which the update is published. If one update is published each second the extra bandwidth needed for the modules are simply their inflation in bytes multiplied by 8.

5.3 Security Management Infrastructure Performance

There are no real-time requirements in the security management infrastructure communication. Here the scalability of latencies is the emphasized property. For the security architecture to be a

viable alternative for a large critical information infrastructure it has to handle large scales.

In the case of the presented security architecture there are two main ways it can scale: increasing the size of the security management plane and by increasing the number of publishers and subscribers. Section 5.3.3 presents experiments on these two aspects, while Section 5.3.2 looks into the performance of the security management communication policy propagation protocol defined in Section 3.3. All the experiments are run on the testbed presented in Section 5.3.1.

5.3.1 Security Management Infrastructure Testbed

All the security management infrastructure performance experiments inherently have different GridStat topologies, but they all are run on the same underlying testbed provided by TCIP [1].

- 4 nodes each with an Intel 2.4 Ghz E6600 dual core CPU with 2 gigabytes of RAM
- Linux Fedora 6 with kernel 2.6.22.7-57
- java2SE 6 version 1.6.0-02
- 100 Mbit Switched LAN

5.3.2 Security Management Communication Policy Propagation

The protocols defined in Section 3.3 are used in order to propagate and activate a change to a security management communication policy. These types of policies always have the narrow scope of only a single communication link from a parent, either an interior-SMS or a leaf-SMS, to a child interior-SMS, leaf-SMS, publisher or subscriber, they thus operate completely independently of the larger topology. Figure 5.2 shows the result of an experiment running 8000 module switches for such a communication link using the `AES` encryption module both as the original module and the new target module.

As the experiments show, the protocol accomplishes the three key changes, the transfer of the 5403 bytes `AES` module and its activation with the new keys, at an average of 25.3 milliseconds.

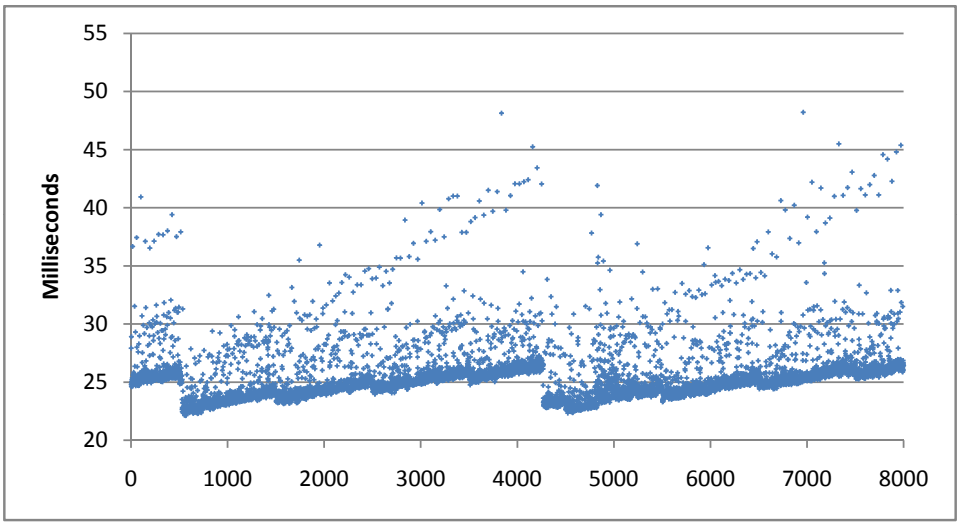

Figure 5.2: Security Management Communication Policy Propagation Latency

This performance is obviously greatly influenced by the speed of the communication link between the parent and the child, but nevertheless shows that latency related to module and key switches over single security management links does not pose any significant limiting factor.

5.3.3 Security Group Policy Propagation

Security group policy propagation is the act of propagating changes done to a security group policy down to publishers and subscribers that have policies belonging to that group. It is thus the main mechanism providing dynamic flexibility for data plane security. The operation consists of the leaf-SMS, that receives a change in a SGP through its user interface from a controller, completing the following tasks:

- Update its publication and subscription policy data base so it adheres to the new SGP.
- Push out the new publication and subscription policies to the affected publishers and subscribers.

When the publishers and subscribers receive the new polices they have to do the following:

- Download the new module(s) if they do not already posses it/them.
- Instantiate and activate the new module set with the new set of keys according to the procedures defined in Section 4.5.

The security group policy propagation latencies are defined as the time it takes from the operator initiating the SGP change until all the publishers and subscribers operate with the new set of modules and keys. While there is no real-time requirement for this type of communication it needs to scale reasonable well to handle a real deployment. The next two sections evaluate this in context of the size of the security management hierarchy and the number of subscribers.

All of the security group policy propagation experiments are done using the `AES` module to provide security management communication security between the different GridStat components.

5.3.3.1 Security Management Plane Hierarchy Scalability

To handle a large scale deployment the security architecture needs to be able to propagate SGP changes with varying number of levels in the security management hierarchy. Two experiments where done for each hierarchy size from one to five levels with the test setups illustrated in Figure 5.3, one without allowing the SMSs to cache the security modules and one with module caching enabled.

Figure 5.4 show how the latency scales with and without module caching enabled when the number of levels in the security management plane increases. Not surprisingly the experiments without module caching have a much steeper increase in latency than ones with caching enabled and it reaches as much as 246 milliseconds for five levels. If we decompose the total latencies into three parts as is shown in Figure 5.5 and 5.6 this becomes even more apparent.

The total latency is the sum of the time it takes to update the policy database at the originating leaf-SMS plus the time it takes to push out the new policies and the time it takes the publishers and subscribers to download and activate the new modules. In both the experiments where the cache is enabled and disabled the time it takes to update the database is constant relative to the

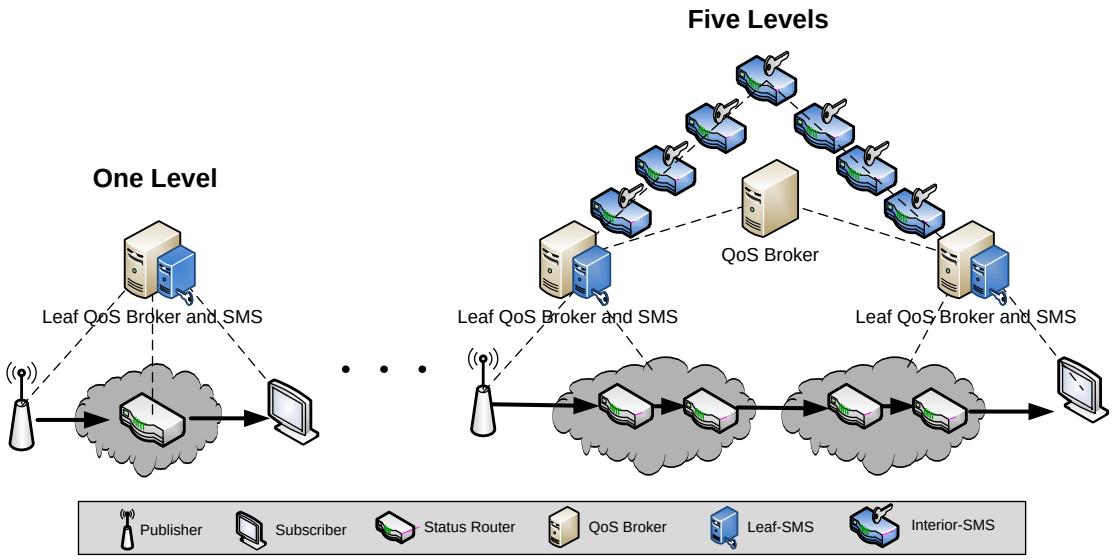

Figure 5.3: Security Group Policy Propagation Hierarchy Scale Test Setup

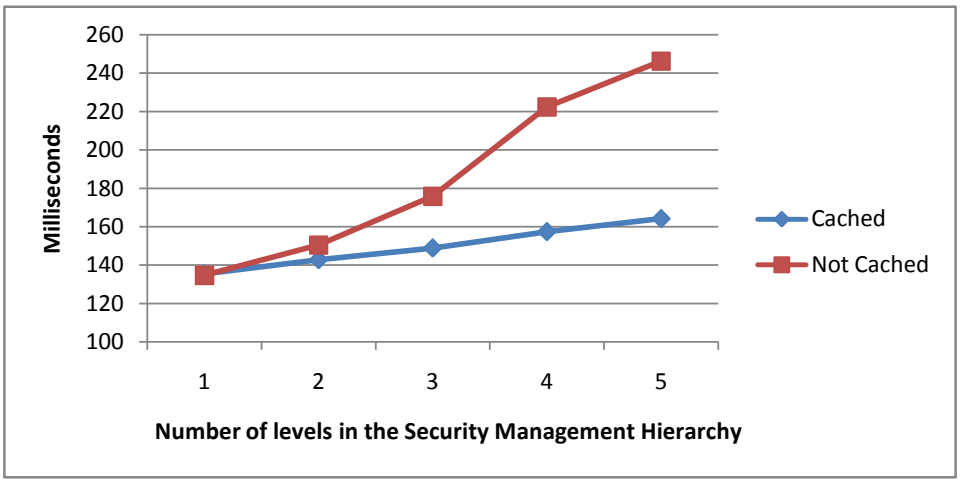

Figure 5.4: Security Group Policy Propagation latency with and without module caching

number of levels in the hierarchy, approximately 72 milliseconds, and thus does not contribute to an latency increase. The time it takes to download the new module, on the other hand, provides the differentiation between enabled and disabled caching. In the cached experiments the time it takes the subscriber to download the module is approximately 50 milliseconds since the module only has to be transferred from the cache of the local leaf-SMS, while in the experiments with caching

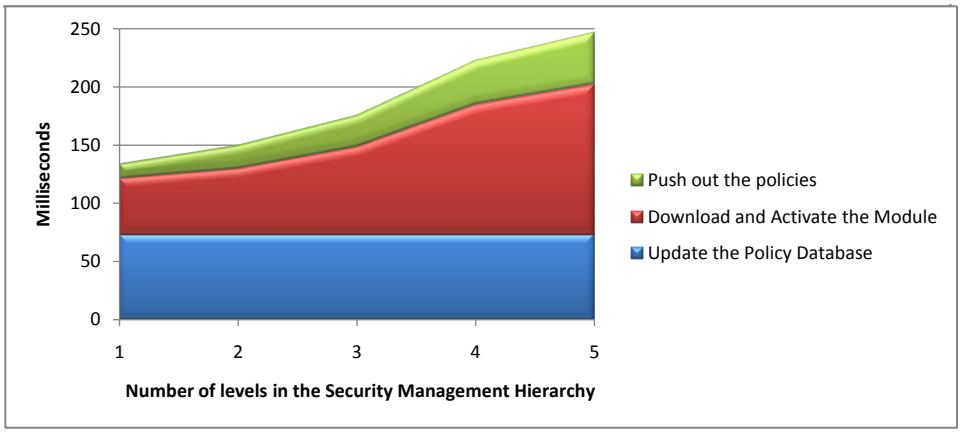

Figure 5.5: Decomposition of the latencies without caching

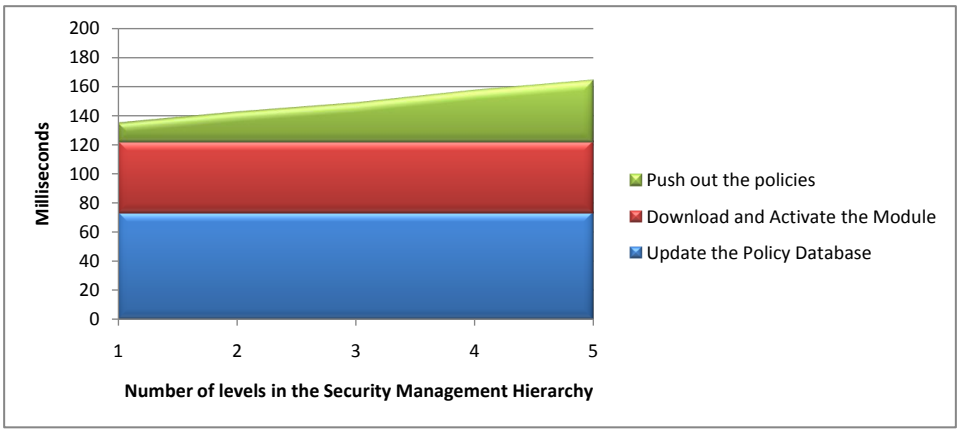

Figure 5.6: Decomposition of the cached latencies

disabled the module has to be routed all the way up and down the hierarchy. As a result the cache disabled experiment with five levels used 82 milliseconds more than the corresponding experiment with caching enabled in accordance with the following analysis:

- Caching Disabled: $U + p(l) + d(l) = TotalLatency$, where U is the constant time it takes to update the policy database, $p(l)$ is the time it takes to push out the new policies, $d(l)$ is the time it takes the subscriber to download and activate the new module and l is the number of levels in the security hierarchy.

- Caching Enabled: $U + p(l) + D = TotalLatency$, where U is the constant time it takes to update the policy database, $p(l)$ is the time it takes to push out the new policies, D is the constant time it takes the subscriber to download and activate the new module and l is the number of levels in the security hierarchy.

The experiments show that with caching enabled the changes to SGPs can still be propagated down to the publishers and subscribers with reasonable large security management planes within a reasonable amount of time.

5.3.3.2 Subscriber scalability

In addition to being able to handle a large security management infrastructure, the architecture has to handle scale with respect to the number of publishers and subscribers that use the infrastructure. To show how the latencies behaved with an increasing number of subscribers the test setup with one through seven subscribers with a five level cache enabled security management plane as depicted in Figure 5.7 was set up.

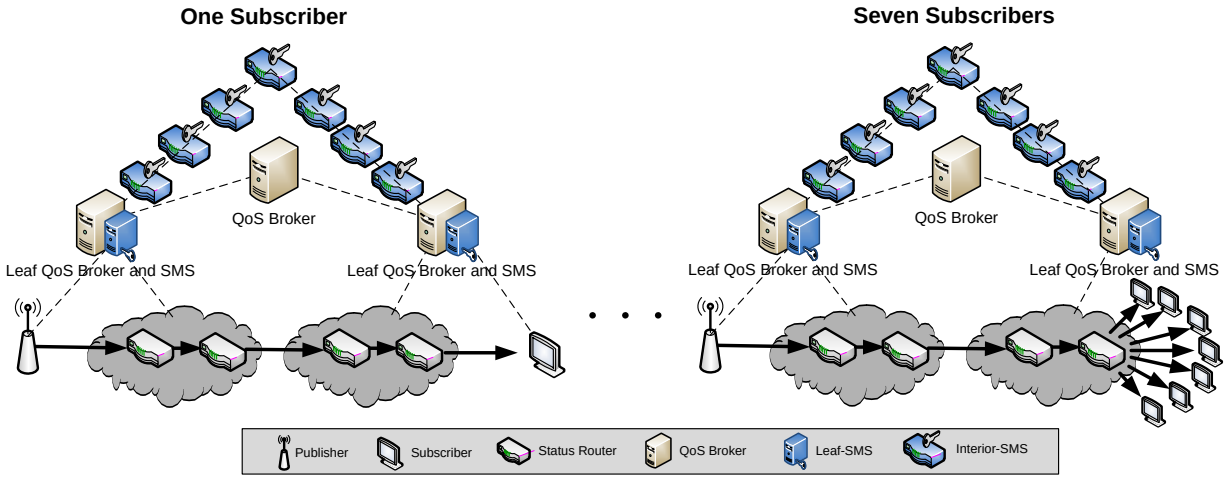

Figure 5.7: Security Group Policy Propagation Subscriber Scale Test Setup

While seven subscribers might look like a small number it is a realistic number in context of the power grid. Because of the sensitivity of the data it is seldom shared indiscriminately. A subscriber

in the local utility's control center, one in their backup center, a couple at oversight organizations and some for their immediate neighbors seems a likely approximation. There are also limitations to how many nodes that can run on the hardware available in the testbed.

The total latency for these experiments can be decomposed into the following formula:

$$U + p(l, s) + MAX(D_i) = TotalLatency$$

Where U is the constant time it takes to update the policy database, $p(l, s)$ is the time it takes to push out the new policies, $MAX(D_i)$ is the time it takes the slowest subscriber to download and activate the new module, l is the number of levels in the security hierarchy and s is the number of subscribers.

The publication and subscription polices are currently being pushed out sequentially and this causes the time it takes push them all out to be the product of the time it takes to push out one policy and the number of polices. As seen in Figure 5.8 this is the main cause of the linear increase in latency when the number of subscribers increase. Future optimizations increasing the parallelism of the policy pushing discussed in Section 6.2.2 have the potential of eliminating the decrease in performance caused by the sequential pushing.

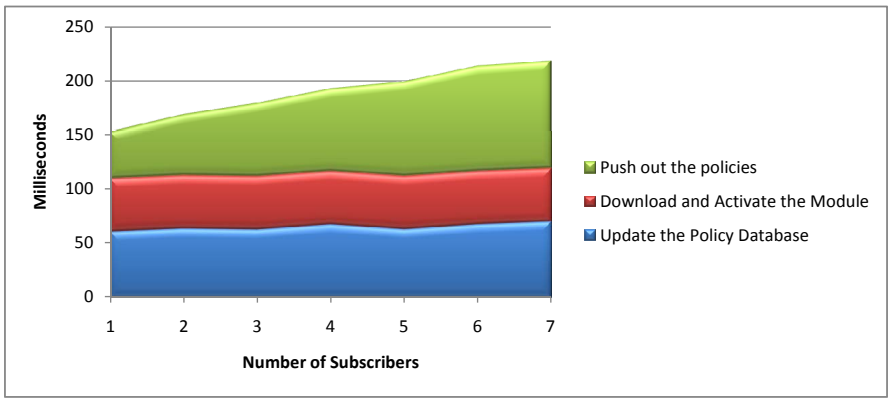

Figure 5.8: Security Group Policy Propagation Subscriber Scale Latencies

Notice that the time used to download and activate a new module is constant with respect to the

number of subscribers. This is an effect of each subscriber downloading the modules in parallel and thus does not impact the overall latency.

As the experiments show there is some room for improvement by achieving a higher degree of parallelism when pushing out new policies, but even without this improvement the security architecture is able to change the security module and keys used for a publisher and seven subscribers within 225 milliseconds.

CHAPTER SIX

CONCLUSIONS AND FUTURE WORK

6.1 Concluding Remarks

The security architecture presented in this thesis addresses the need for flexible security critical information systems with long lifecycles. The current information system for the power grid is used as an example since it was designed several decades ago and does not take advantage of the developments in distributed systems that have happened since then. The grid is constantly being operated closer to its limits and as a result the increasingly richer monitoring information becomes more and more critical. GridStat provides a QoS-managed publish-subscribe framework that addresses many of the issues that current hard wired, strictly hierarchical information system does not, but have lacked an implemented security architecture.

This thesis explores the conventional industry approaches of using PKI based techniques, such as TLS/SSL, to provide the needed security mechanisms for GridStat and finds that they do not provide the flexibility that is needed. Information systems for the power grid are huge distributed systems of largely unmanned nodes with 25 year life expectancy. To address the need for flexibility that can handle changes in the security field, which is in a constant arms race between the security scientists and the hacker community, an alternative novel security architecture based on the use of transparent interchangeable security modules is presented based on the presented threat model. This security architecture, extension to GridStat's existing management plane, is called the security management plane (SMP). It contains a dynamic module repository with security modules that can be assigned and reassigned to publishers and subscribers at a status variable granularity. This means that each status variable may be assigned its own set of security modules providing the optimal tradeoffs between different security aspects such as encryption, obfuscation, authentication, integrity and filtering, and performance properties such as latency.

To support the dynamic module approach in the data plane the security mechanisms securing the security infrastructure needs the same level of flexibility. There is little to be gained from building something really flexible on top of a static foundation since that would expose the overlying dynamic building to any cracks found in the static foundation. To address this a protocol was developed that takes advantage of the relatively static attributes of information systems in the power grid such as its constant set of services and the fact that every component that wants to connect to the network needs to be explicitly allowed. This is done by leveraging pre-loading of keys to achieve complete flexibility in aspect to the use of security algorithms.

In addition to presenting the design and implementation of the security architecture the thesis explores the different types of relevant security modules, alternative approaches and algorithms for each type of security module and a series of implemented proof-of-concept modules based on this exploration. The evaluation of the security architecture with the implemented proof-of-concept modules showed that the modular security architecture was able to provide confidentiality, integrity, and authentication well within the performance requirements set by the power grid industry.

The evaluation of the security architecture infrastructure also showed that it is able to handle large scale systems. With five levels of security management servers in the security management plane hierarchy the over-the-wire replacement of a module used to secure a average publication and its subscriptions takes less than 200 milliseconds.

Overall this thesis has presented and evaluated a novel security architecture for GridStat. It was shown that, even though there exists many aspects that needs to be explored deeper through future research, the security architecture provides GridStat with the mechanisms needed to support rich and flexible security for its data plane that performs within the real-time requirement put on it by the power grid.

6.2 Future Work

6.2.1 Implement Optimized Modules

The proof-of-concept modules where implemented to show that it was feasible to provide the needed security through the use of combinations of modules but there is a lot of room for improvement. First of all a from-the-ground-up implementation of the different algorithms would have removed the need to translate the GridStat formatted data to a form that Sun's JCE can understand and back again. This translation adds two unnecessary copy operations that contribute greatly to the total latency, especially for larger data samples.

Secondly if GridStat is going stay mainly Java based there has to be put more work into solving the garbage collection problem. Having a garbage collector that could kick in at any time is in conflict with achieving real-time guarantees. This is not just something that has to be solved for the parts of the security architecture that have real-time requirements, but also every other GridStat component that are involved in data delivery.

6.2.2 Increase the parallelism in the Security Management plane

Currently the process of pushing out updated polices from the security management plane is done sequentially. Policy number i is never pushed out before it is confirmed that $i-1$ has been received by the intended publisher or subscriber. This amplifies, as the experimental results in Section 5.3.3.1 show, any latencies caused by routing policies through the security hierarchy since each of the policies then need to be encrypted independently.

By increasing the parallelism of the policy propagation this could be partially remedied. Every SMS has a set of children and a single parent. Instead of letting the SMS sequentially push out the policies without considering whether there is more than one policy that is going to the same child or parent, the policies could be bundled according to their immediate target. This reduces the number of transfers needed by only sending one message per link, thus providing a higher degree of parallelism and potentially increase the performance.

6.2.3 Deployment and independent evaluation

Before any security architecture can be deployed it needs to be independently tested. It is not enough to show that the architecture is theoretically sound. That says little about the implementation of the architecture. Therefore there is a project already underway between the GridStat group, Avista Utilities, Pacific Northwest National Laboratory (PNNL) and Idaho National Laboratory (INL) where a small GridStat setup with the security extension will be set up for evaluation. INL will employ a *red team* trying to find weaknesses and/or holes in the security implementation.

The results from this project will then be fed back into the development of GridStat as a whole and the security architecture in particular in order to continuously improve its capabilities.

6.2.4 Policy Development

The security extension to GridStat provides the mechanisms needed to provide flexible security for its data plane, but future work has to be done to develop policies that take advantage of these mechanisms. Since the security extension is capable of per-status-variable granularity more research should be put into classifying the different types of status variables and their different security and performance needs. This would enable the development of policies that utilize the possibility of combining different modules to achieve the best trade off between the different aspects of security and performance for each group of status variables. There are many possible combinations such as status variables that are very sensitive, but might not have strict latency requirements, status variables that are not especially sensitive but have strict integrity requirements, while others again might need a little of everything.

Developing an optimal set of security polices is a job that has to be done in tight cooperation with the industry over a extended period of time to achieve the best result.

6.2.5 Develop Security Mechanisms for the QoS-Broker management

This thesis has focused on securing GridStat data plane in an effort to get a manageable problem space for a single thesis. Future work has to be undertaken to extend the security architecture

to the QoS broker hierarchy. Worth noting is that even without securing the QoS hierarchy the end-to-end approach, combined with the access control to the keys and modules in the security management plane, makes it impossible for an attacker to gain access to any published data through the unsecured QoS hierarchy. The weakness is that attackers easily can use the unsecured QoS broker hierarchy to disrupt legal subscribers from getting their data and thus effect denial of service.

Extending the security architecture to the general management plane can be done in several ways. One way is to let the general management communications use the recently developed RPC scheme Ratatoskr [35] that is built on top of the data plane. Since the data plane is secured by the security architecture presented in this thesis this would transitively also secure the management plane.

Alternatively a scheme similar to the one used to secure the security infrastructure, based on sets of preloaded keys, could be adopted. This is a somewhat more cumbersome approach, but has the advantage of not needing to route the management communication through the data plane. Finally a completely independent architecture could be developed and put in place.

6.2.6 End Point Security

The threat model for presented security architecture does not include end point attacks. These types of attacks are directed at the end point in the system, either through local physical access or remote connections. Future work has to be undertaken to find the best way to secure the end points and this is especially important in the power grid since many of the end points are unmanned. Defenses against un-authorized physical access have to be in place before a real deployment of GridStat can be undertaken.

6.2.7 Publication Security

The presented security architecture provides access control by the controlling which subscribers get access to the modules and keys securing the different publications, but this is currently enforced through simplistic static access control lists. As Pesonon et. al. observe in [29] any system that

works across security domains needs more flexible approaches.

To address the need for a more flexible access control there is currently ongoing work with another member of the TCIP project, Adam Lee, to integrate the attribute based trust scheme of *TrustBuilder* [26]. The integration allows potential subscribers to incrementally build chains of trust with the security management plane. Keeping access control lists updated with every operator across several utilities is at best cumbersome, TrustBuilder enables GridStat to avoid that problem by authenticating users based on their attributes instead of explicit identification.

The integration of TrustBuilder2 into the security architecture is being done through a callback interface as shown in Figure 6.1. The utility worker can specify the credential needed to access the publication he/she sets up by supplying a *trust builder policy*. This trust builder policy is then embedded into the the publication policy that is sent to the SMP. Whenever a potential subscriber tries to subscribe to that publication the security management server feeds the TrustBuilder policy into its embedded TrustBuilder server. The TrustBuilder server then initiates an incremental trust negotiation with the embedded TrustBuilder client embedded in the subscriber. If the subscriber's credentials fulfill the publication's requirements the TrustBuilder returns true and the modules and keys are sent to the subscriber.

TrustBuilder allows the security architecture to provide GridStat with dynamic attribute based publication security. This makes makes it much more manageable to enforce access control across security domains. A demonstration of the integration, limited to one level in the security management plane, was shown at the TCIP projects annual National Science Foundation (NSF) review.

6.2.8 Intrusion Detection

The security architecture needs to be complemented by a intrusion detection system (IDS) that can raise alarms when irregularities are discovered. While the security architecture provides several hooks for detecting irregularities, it does not act on them. An example of such a hook is the integrity modules. They check the integrity of incoming data, but only throws an exception if an

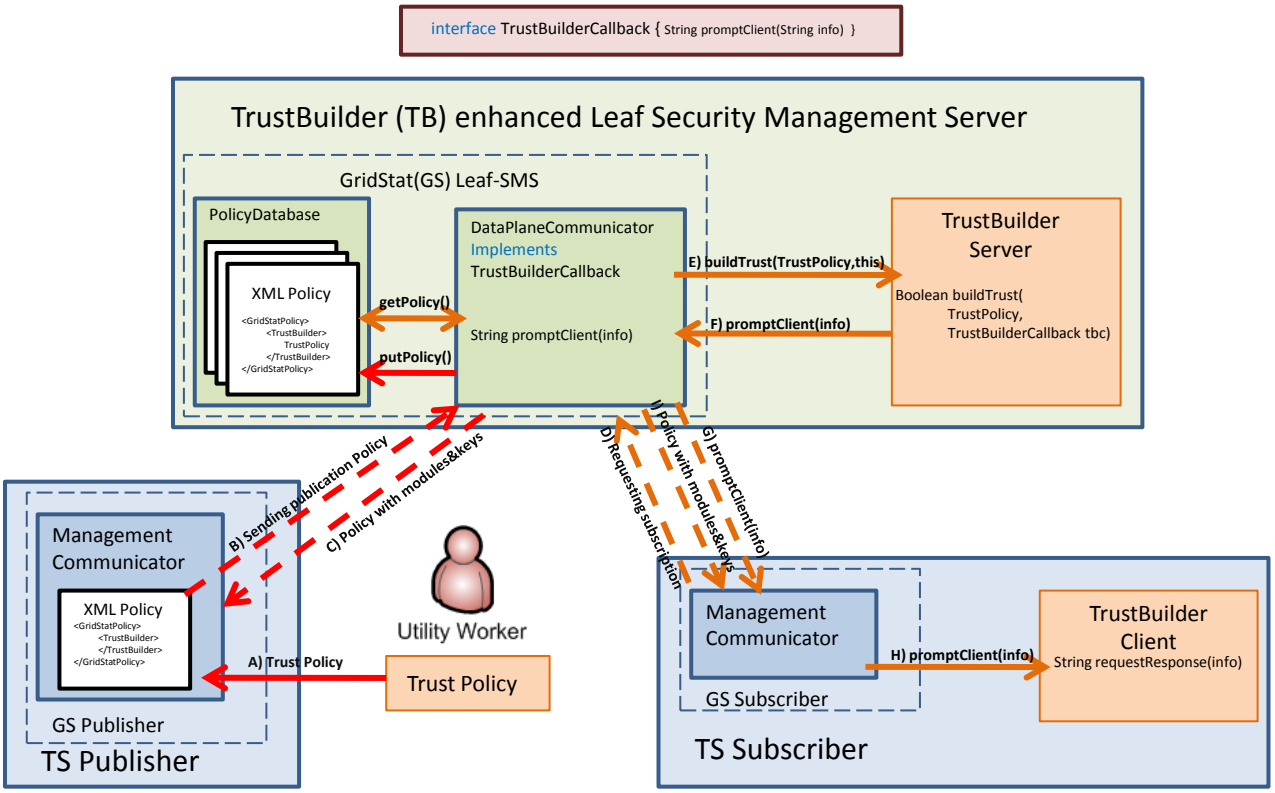

Figure 6.1: TrustBuilder2 integration

irregularity is discovered. An IDS could use such information, together with other data such as failed re-keying and module attempts, to provide more intelligent handling of such events.

BIBLIOGRAPHY

[1] TCIP: Trustworthy Cyber Infrastructure for the Power Grid
URL: http://www.iti.uiuc.edu/tcip/index.html.

[2] Microsoft TechNet. Certificate Life Cycle,
URL: http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/distrib/dscj_mcs_evyy.mspx?mfr=true Last Checked: 2007-10-02.

[3] ITU Telecommunication Standardization Sector (ITI-T). ITU-T RECOMMENDATION X.509, July 2005, URL: http://www.itu.int/rec/T-REC-X.509-200508-I/en.

[4] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). Federal Information Processing Standards Publication (FIPS PUB) 186-3, March 2006.

[5] National Institute of Standards and Technology (NIST). Secure Hash Signature Standard (SHS). Federal Information Processing Standards Publication (FIPS PUB) 180-2, August 2002.

[6] National Institute of Standards and Technology (NIST). Data Encryption Standard (DES). Federal Information Processing Standards Publication (FIPS PUB) 46-3, October 1999.

[7] National Institute of Standards and Technology (NIST). Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher. NIST Special Publication (NIST SP) 800-67, May 2004.

[8] Sun Microsystems. Java SE Security, Sun Developer Network
URL: http://java.sun.com/javase/technologies/security Last checed 10.25.07.

[9] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS PUB) 197, Nov 2001.

[10] S. F. Abelsen. Adaptive gridstat information flow mechanisms and management for power grid contingencies. Master's thesis, Washington State University, August 2007.

[11] C. Adams and F. S. RFC 4210: Internet x.509 public key infrastructure certificate management protocol, September 2005.
URL: http://tools.ietf.org/html/rfc4210.

[12] D. E. Bakken, C. H. Hauser, H. Gjermundrod, and A. Bose. Towards more flexible and robust data delivery for monitoring and control of the electric power grid. Technical Report 009, Washington State University, May 2007.

[13] D. E. Bakken, R. Parameswaran, D. M. Blough, A. A. Franz, and T. J. Palmer. Data obfuscation: Anonymity and desensitization of usable data sets. *IEEE Security and Privacy*, 2(6):34–41, 2004.

[14] R. Bragg, M. Rhodes-Ousley, K. Strassberg, and et al. *Network Security, The Complete Reference*. McGraw-Hill/Osborne, 2004. Chapter 7.

[15] F. Cleveland. IEC TC57 security standards for the power systems information infrastructure beyond simple encryption.
URL: http://xanthus-consulting.com/Publications/, June 2007. IEC TC57 WG15 Security Standards White Paper ver 11.

[16] T. Dierks and E. Rescorla. RFC 4346: The transport layer security (tls) protocol version 1.1, April 2006. URL: http://tools.ietf.org/html/rfc4346.

[17] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[18] I. Dionysiou, K. H. Gjermundrd, and D. Bakken. Fault tolerance issues in publish-subscribe status dissemination middleware for the electric power grid. In *Supplement of the 2002 International Conference on Dependable Systems and Networks*. IEEE/IFIP, June 2002.

[19] C. Ellison and B. Schneier. Ten risks of pki: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[20] G. Garon and R. Outerbridge. DES watch: an examination of the sufficiency of the data encryption standard for financial institution information security in the 1990s. *SIGSAC Rev.*, 9(4):29–45, 1991.

[21] K. H. Gjermundrod. *Flexible QoS-Managed Status Dissemination Middleware Framework for the Power Grid*. PhD thesis, Washington State University, August 2006.

[22] C. H. Hauser, D. E. Bakken, and A. Bose. A failure to communicate: next generation communication requirements, technologies, and architecture for the electric power grid. *Power and Energy Magazine, IEEE*, 3(2):47–55, April 2005.

[23] K. Kaukonen and R.Thayer. INTERNET-DRAFT: A stream cipher encryption algorithm "arcfour", July 1999. URL:
http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt.

[24] H. Khurana. Scalable security and accounting services for content-based publish/subscribe systems. In H. Haddad, L. M. Liebrock, A. Omicini, and R. L. Wainwright, editors, *SAC*, pages 801–807. ACM, 2005.

[25] X. Lai and J. L. Massey. A proposal for a new block encryption standard. *Lecture Notes in Computer Science*, 473:389–402, 1991.

[26] A. J. Lee. Trustbuilder2 user manual, May 2007. Department of Computer Science Univ. of Illinois at Urbana-Champaign and Sandia National Laboratories.

[27] P. Mockapetris. RFC 1035: Domain names - implementation and specification, November 1987. URL: http://tools.ietf.org/html/rfc1035.

[28] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[29] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. Encryption-enforced access control in dynamic multi-domain publish/subscribe networks. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 104–115, New York, NY, USA, 2007. ACM Press.

[30] C. Raiciu and D. Rosenblum. Enabling Confidentiality in Content-Based Publish/Subscribe Infrastructures. In *Proc. Second IEEE Communications Society/CreateNet Int. Conf. on Security and Privacy in Communication Networks (SecureComm 2006)*, Aug.-Sep. 2006.

[31] R. Rivest. RFC 1321: The md5 message-digest algorithm, April 1992. MIT Laboratory for Computer Science and RSA Data Security, Inc.

[32] R. Rivest. RFC 2268: A description of the rc2(r) encryption algorithm, March 1998. URL: http://tools.ietf.org/html/rfc2268.

[33] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 26(1):96–99, 1983.

[34] B. Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, December.

[35] E. S. Viddal. Ratatoskr: Wide-area actuator rpc over gridstat with timeliness, redundancy and safety. Master's thesis, Washington State University, December 2007.

[36] C. Wang, A. Carzaniga, D. Evans, and A. Wolf. Security issues and requirements for internet-scale publish-subscribe systems. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*, page 303, Washington, DC, USA, 2002. IEEE Computer Society.

[37] J. Weiss. *Java Cryptography Extensions: Practical Guide for Programmers*. Morgan Kaufmann, 1 edition edition, 2004.

[38] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. Negotiating trust in the web. *Internet Computing, IEEE*, 6:30–37, December 2002.

APPENDIX

APPENDIX ONE
UNFILTERED EXPERIMENT RESULTS

As mentioned in Chapter 5 the experimental result presented is filtered for the interruptions from the Java garbage collector (GC). This is the case for both the data plane and some of the hierarchical infrastructure experiments. In an effort to illuminate all aspects of the performance Section A.1 and A.2 presents the unfiltered results.

A.1 Unfiltered Module Experiments

Table A.1 and A.2 contain the filtered experimental results from the publisher and subscriber that the evaluation in Chapter 5 is rooted on, while Table A.3 and A.4 in comparison present the unfiltered results. By comparing the two one find that there is a trivial increase in latency without the filter that does not have a significant overall impact, but that the filter has a huge impact on the standard deviation. This indicates that if there could be a way to control when and how the garbage collector runs as is discussed in Section 6.2.1, the latency it adds would be tolerable.

Figure A.1 and Figure A.2 is an example of what the result of the GC filtering. Figure A.1 presents the each of the unfiltered results from applying the *DES* module on 10,000 status updates of 200 bytes, while Figure A.2 shows the same results filtered for GC interruptions. Notice that the scale on Figure A.1 is logarithmic.

A.2 Unfiltered Infrastructure Experiments

Since the infrastructure performance does not have any real-time requirements the jitter in the performance caused by the GC does not have the same consequence as in the data plane. In addition the performance of the infrastructure is measured in milliseconds instead of microseconds and as a result it is only in cases where a significant amount of memory allocation and de-allocation that the GC filtering has any real impact. This becomes apparent when the filtered and unfiltered latencies of SGP propagation experiments with the disabled are compared in Figure A.3 and A.4. By decomposing the total latencies of the cached experiments into subparts, see Figure A.5 and A.6, we can see that the increased latency is caused by the memory allocation and de-allocation needed to download and instantiate the module.

	Integer			String with 200 characters		
Encryption	**Average**	**Median**	**Std Deviation**	**Average**	**Median**	**Std Deviation**
CaesarCipher	2,856	2,794	743	3,295	3,277	579
BLOWFISH	7,336	6,858	2,169	16,091	14,905	4,053
AES	8,341	7,818	2,740	16,203	15,306	2,988
DES	8,059	7,496	2,618	22,945	22,071	3,360
TripleDES	9,748	9,078	3,211	43,642	42,304	4,448
Obfuscation						
OneTimePadObf	8,303	7,069	3,506	19,917	18,701	4,152
AESObfuscation	52,804	51,169	16,873	57,260	59,346	7,783
Integrity						
CRC	2,589	2,527	671	16,789	16,770	1,306
MD5ErrorCheck	5,187	4,943	1,685	7,721	7,121	2,239
SHAErrorCheck	6,490	5,859	2,808	9,823	9,356	2,366
SHA512ErrorCheck	11,051	10,378	2,755	17,023	16,083	3,240
Authentication						
SimpleAuth	2,161	2,130	581	2,134	2,111	580
RSA	114,002,572	112,183,055	3,826,042	114,858,257	112,884,239	3,811,959
Combinations						
MD5/Blowfish	16,612	16,409	2,412	27,208	24,423	6,827
Blowfish/MD5	12,746	13,698	2,909	24,588	24,321	2,692
OneTimePadObf/Blowfish/MD5	14,042	13,347	3,473	54,404	55,424	6,677
OneTimePadObf/AES/SHA	15,376	15,386	2,324	60,701	62,140	8,195
SimpleAuth/OneTimePadObf/Blowfish	13,393	11,438	4,156	46,717	47,756	5,256
SimpleAuth/OneTimePadObf/Blowfish/SHA	15,765	15,712	2,311	59,583	60,798	6,299
SimpleAuth/OneTimePadObf/Crc/Blowfish	15,068	14,977	1,337	121,353	121,783	6,357
AESObfuscation/TripleDES/SHA512	70,602	69,799	11,136	135,048	136,378	18,092

Table A.1: Publisher side latency in nanoseconds after filtering out the GC

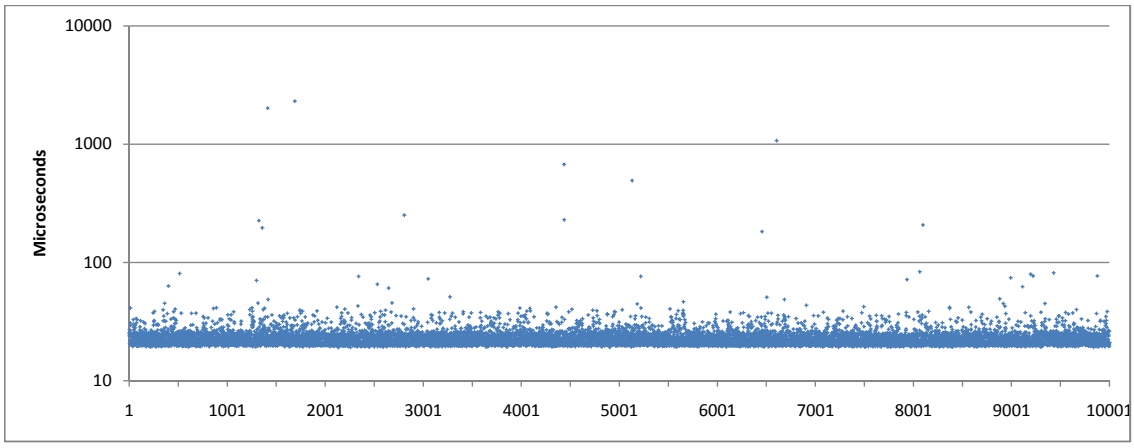

Figure A.1: Unfiltered publisher side DES module latency

	Integer			String with 200 characters		
Encryption	**Average**	**Median**	**Std Deviation**	**Average**	**Median**	**Std Deviation**
CaesarCipher	3,224	3,289	950	4,844	5,006	1,084
BLOWFISH	9,318	8,239	3,542	32,263	33,837	6,361
AES	8,101	8,036	2,362	33,231	34,576	7,047
DES	9,561	8,546	3,624	42,486	39,151	13,269
TripleDES	11,218	10,057	4,194	70,439	75,211	11,836
Obfuscation						
OneTimePadObf	7,489	6,465	3,088	28,042	29,765	5,096
AESObfuscation	34,134	34,672	8,277	58,800	61,424	16,232
Integrity						
CRC	2,654	2,715	631	17,075	17,699	2,486
MD5ErrorCheck	4,815	4,822	1,543	6,804	7,091	1,499
SHAErrorCheck	5,938	6,071	1,663	9,423	9,899	1,918
SHA512ErrorCheck	12,481	13,207	2,717	19,028	20,132	3,588
Authentication						
SimpleAuth	2,332	2,393	688	2,414	2,494	642
RSA	933,402	930,034	148,468	944,225	938,888	155,471
Combinations						
MD5/Blowfish	11,844	12,645	2,688	35,231	37,746	7,593
Blowfish/MD5	10,412	9,929	2,400	34,434	36,513	7,450
OneTimePadObf/Blowfish/MD5	14,736	15,003	3,103	52,885	50,485	13,367
OneTimePadObf/AES/SHA	15,946	16,342	3,169	60,111	57,640	15,462
SimpleAuth/OneTimePadObf/Blowfish	11,392	11,528	3,116	39,115	41,137	6,251
SimpleAuth/OneTimePadObf/Blowfish/SHA	16,577	16,177	4,211	49,986	52,751	9,272
SimpleAuth/OneTimePadObf/Crc/Blowfish	13,280	13,207	2,494	96,782	97,811	10,885
AESObfuscation/TripleDES/SHA512	59,304	61,585	13,099	108,073	107,830	15,894

Table A.2: Subscriber side latency in nanoseconds after filtering out the GC

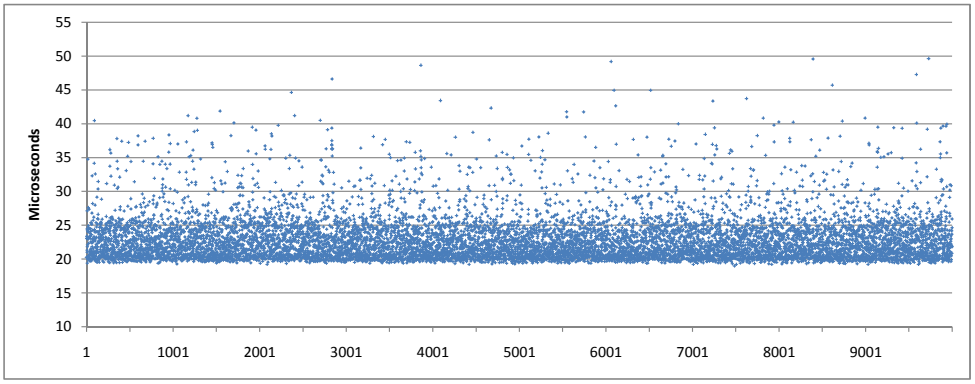

Figure A.2: Unfiltered publisher side DES module latency

	Integer			String with 200 characters		
Encryption	**Average**	**Median**	**Std Deviation**	**Average**	**Median**	**Std Deviation**
CaesarCipher	2,856	2,794	743	3,295	3,277	579
BLOWFISH	7,419	6,866	4,505	17,071	14,841	57,397
AES	8,555	7,821	7,862	17,953	15,366	123,886
DES	8,434	7,477	21,258	23,846	22,127	33,664
TripleDES	12,052	9,089	209,932	45,337	42,233	52,567
Obfuscation						
OneTimePadObf	8,303	7,069	3,506	23,524	18,682	92,371
AESObfuscation	59,957	51,169	230,681	57,895	59,375	57,310
Integrity						
CRC	2,589	2,527	671	16,789	16,770	1,306
MD5ErrorCheck	6,582	4,942	105,655	7,868	7,121	6,468
SHAErrorCheck	6,990	5,865	33,502	10,700	9,363	49,496
SHA512ErrorCheck	14,260	10,386	100,219	17,698	16,084	24,036
Authentication						
SimpleAuth	2,161	2,130	581	2,197	2,187	943
RSA	114,002,572	112,183,055	4,394,433	114,858,257	112,884,239	4,735,562
Combinations						
MD5/Blowfish	16,600	16,327	2,556	27,457	24,397	25,873
Blowfish/MD5	12,748	13,698	2,953	24,610	24,322	2,777
OneTimePadObf/Blowfish/MD5	17,820	15,570	18,984	54,684	55,323	35,147
OneTimePadObf/AES/SHA	19,636	16,132	12,736	70,601	62,215	335,739
SimpleAuth/OneTimePadObf/Blowfish	14,992	12,577	7,431	48,992	47,707	34,585
SimpleAuth/OneTimePadObf/Blowfish/SHA	15,915	15,705	13,012	59,802	60,663	17,145
SimpleAuth/OneTimePadObf/Crc/Blowfish	15,495	14,977	42,763	128,674	121,783	6,357
AESObfuscation/TripleDES/SHA512	73,414	69,640	185,838	142,662	136,696	220,490

Table A.3: Publisher side latency in nanoseconds without GC filtering

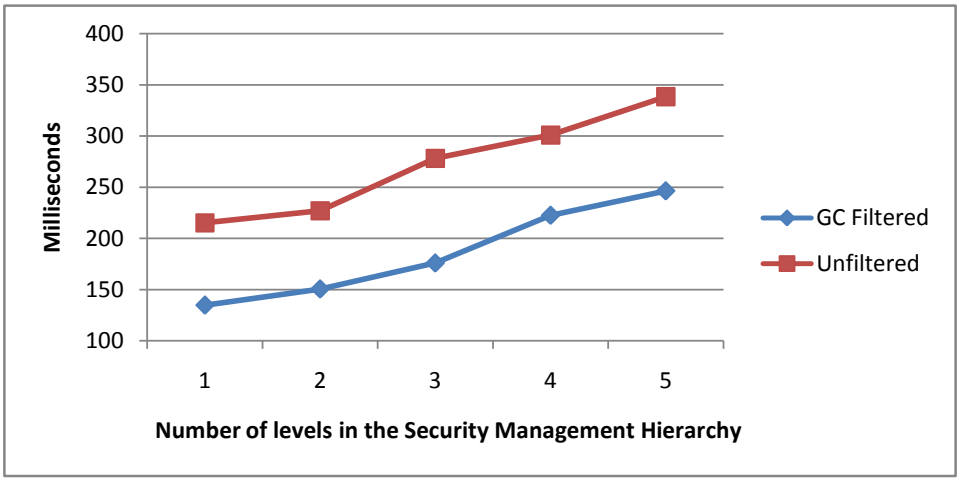

Figure A.3: GC filter impact on SGP propagation latency with module caching disabled

	Integer			String with 200 characters		
Encryption	**Average**	**Median**	**Std Deviation**	**Average**	**Median**	**Std Deviation**
CaesarCipher	3,550	3,289	20,046	5,062	5,006	7,560
BLOWFISH	9,944	8,239	18,145	34,324	33,814	47,967
AES	8,508	8,040	15,706	34,410	34,594	38,179
DES	9,985	8,564	13,742	43,328	39,144	16,330
TripleDES	12,473	10,061	33,641	71,170	75,207	29,871
Obfuscation						
OneTimePadObf	8,030	6,469	19,732	28,952	29,847	11,282
AESObfuscation	36,514	34,672	80,595	77,110	61,645	1,147,388
Integrity						
CRC	2,654	2,715	631	17,093	17,703	2,887
MD5ErrorCheck	5,820	4,822	41,807	7,774	7,027	14,505
SHAErrorCheck	6,645	6,071	51,796	9,796	9,891	15,535
SHA512ErrorCheck	13,561	13,211	54,528	25,239	20,129	556,052
Authentication						
SimpleAuth	2,468	2,393	5,415	28,952	29,847	11,282
RSA	942,645	902,598	902,407	948,177	938,885	256,731
Combinations						
MD5/Blowfish	11,915	12,648	4,915	35,786	37,739	30,293
Blowfish/MD5	10,970	9,929	31,805	35,067	36,531	15,915
OneTimePadObf/Blowfish/MD5	15,416	15,004	33,374	59,630	50,497	340,033
CaesarObfuscation/AES/SHA	16,906	16,345	37,966	65,884	57,543	316,650
SimpleAuth/OneTimePadObf/Blowfish	11,507	11,528	5,860	43,508	41,073	392,590
SimpleAuth/OneTimePadObf/Blowfish/SHA	17,173	16,181	20,888	54,050	52,814	270,225
SimpleAuth/OneTimePadObf/Crc/Blowfish	13,285	13,207	2,529	104,439	98,272	200,103
AESObfuscation/TripleDES/SHA512	65,274	61,644	190,536	112,115	107,694	138,261

Table A.4: Subscriber side latency in nanoseconds without GC filtering

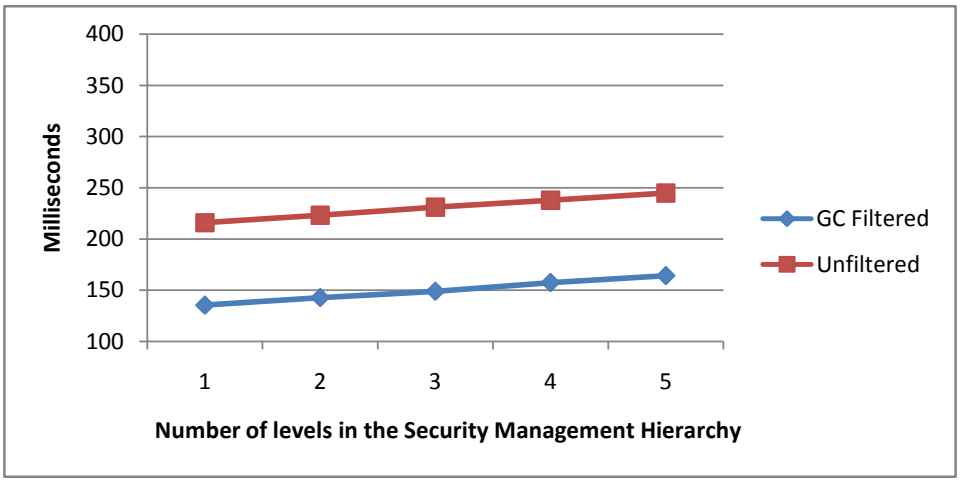

Figure A.4: GC filter impact on SGP propagation latency with module caching

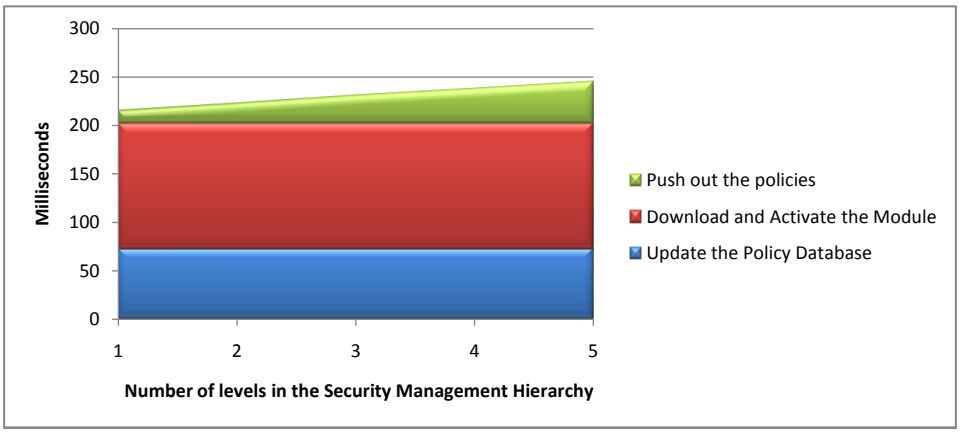

Figure A.5: SGP propagation latency decomposition without GC filtering and caching enabled

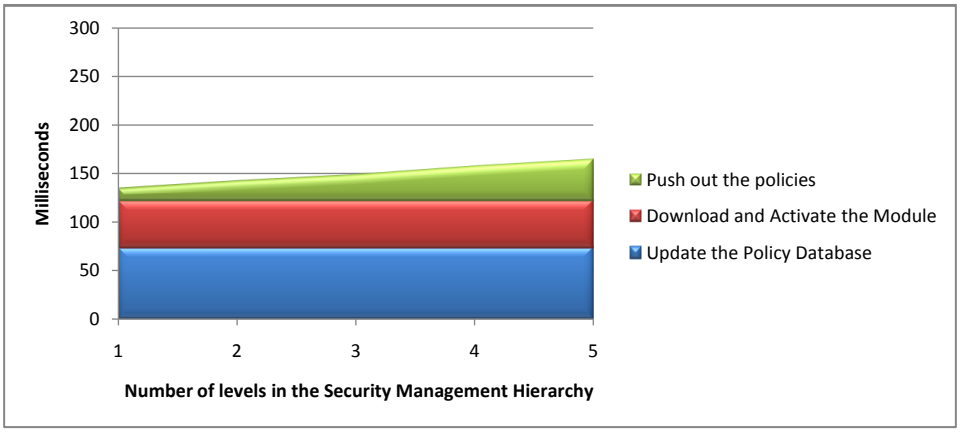

Figure A.6: SGP propagation latency decomposition with module caching and GC filtering