DEGREE PROJECT IN COMMUNICATION SYSTEMS, SECOND LEVEL
STOCKHOLM, SWEDEN 2014

Hardware Security Module Performance Optimization by Using a “Key Pool”

Generating keys when the load is low and saving in the external storage to use when the load is high

NIMA SABOONCHI

KTH ROYAL INSTITUTE OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY

Hardware Security Module Performance Optimization by Using a “Key Pool”

Generating keys when the load is low and saving in the external storage to use when the load is high

Nima Saboonchi

2014-12-25

Master’s Thesis

Examiner and Academic adviser
Gerald Q. Maguire Jr.

Industrial adviser
Roberth Lundin

KTH Royal Institute of Technology
School of Information and Communication Technology (ICT)
Department of Communication Systems
SE-100 44 Stockholm, Sweden

Abstract

This thesis project examines the performance limitations of Hardware Security Module (HSM) devices with respect to fulfilling the needs of security services in a rapidly growing security market in a cost-effective way. In particular, the needs due to the introduction of a new electronic ID system in Sweden (the Federation of Swedish eID) and how signatures are created and managed..

SafeNet Luna SA 1700 is a high performance HSM's available in the current market. In this thesis the Luna SA 1700 capabilities are stated and a comprehensive analysis of its performance shows a performance gap between what HSMs are currently able to do and what they need to do to address the expected demands. A case study focused on new security services needed to address Sweden's e-Identification organization is presented. Based upon the expected performance demands, this thesis project proposes an optimized HSM solution to address the identified performance gap between what is required and what current HSMs can provide. A series of tests were conducted to measure an existing HSM's performance. An analysis of these measurements was used to optimize a proposed solution for selected HSM or similar HSMs. One of the main requirements of the new signing service is the capability to perform fifty digital signatures within the acceptable response time which is 300 ms during normal hours and 3000 ms during peak hours. The proposed solution enables the HSM to meet the expected demands of 50 signing request per second in the assumed two hours of peak rate at a cost that is 1/9 of the cost of simply scaling up the number of HSMs.

The target audience of this thesis project is Security Service Providers who use HSMs and need a high volume of key generation and storing. Also HSM vendors consider this solution and add similar functionality to their devices in order to meet the desired demands and to ensure a better future in this very rapidly growing market.

Key Words

HSM, Digital Signature, PKI, e-Identification, RSA, SAML

Sammanfattning

Detta examensarbete undersöker prestandabegränsningar för Hardware Security Module (HSM) enheter med avseende på att uppfylla behov av säkerhetstjänster i en snabbt växande marknad och på ett kostnadseffektivt sätt. I synnerhet på grund av de säkerhetskrav som nu existerar/tillkommit efter införandet av ett nytt elektroniskt ID-system i Sverige (Federationen för Svensk eID) och hur underskrifter skapas och hanteras.

SafeNet Luna SA 1700 är en högpresterande HSM enhet tillgänglig på marknaden. I den här avhandlingen presenteras nuvarande HSM kapacitet och en omfattande analys av resultatet visar ett prestanda gap mellan vad HSMS för närvarande kan göra och vad som behöver förbättras för att ta itu med de förväntade kraven.

En fallstudie fokuserad på nya säkerhetstjänster som krävs i och med Sveriges nya e-Identifiering presenteras. Baserat på resultatet i den här avhandlingen föreslås en optimerad HSM lösning för att tillgodose prestanda gapet mellan vad HSM presterar och de nya krav som ställs.

Ett flertal tester genomfördes för att mäta en befintlig HSM prestanda. En analys av dessa mätningar användes för att föreslå en optimerad lösning för HSMS (eller liknande) enheter. Ett av de huvudsakliga kraven för den nya signeringstjänsten är att ha en kapacitet av 50 digitala signaturer inom en accepterad svarstidsintervall, vilket är 300ms vid ordinarie trafik och 3000ms vid högtrafik. Förslagen i avhandlingen möjliggör HSM enheten att tillgodose kraven på 50 signeringen per sekund under två timmars högtrafik, och till en 1/9 kostnad genom att skala upp antalet HSMs.

Målgruppen i den här avhandlingen är användare av HSMs och där behovet av lagring och generering av nycklar i höga volymer är stort. Även HSM leverantörer som kan implementera den här optimeringen/lösningen i befintlig funktionalitet för att tillgodose det här behovet i en alltmer växande marknad.

Nyckelord

HSM, Digitala signatur, PKI, e-legitimation, RSA, SAML

Table of Contents

Abstract i
Key Words i
Sammanfattning iii
Nyckelord iii
Table of Contents v
List of Figures vii
List of Tables ix
List of Acronyms and Abbreviations xi
Acknowledgements xiii
1 Introduction 1
1.1 Problem definition 1
1.2 New e-ID System 1
1.3 Signature Service requirements by large Authorities 2
1.4 Plugin problem 2
1.5 Purpose 3
1.6 Advantages of the new service model 4
1.7 Proof and validation of a signature in new system 5
1.8 Service Design 5
1.9 Bottleneck 7
1.10 Goal 7
1.11 Aim of this master thesis 8
1.12 Research Methodology 8
1.13 Delimitations 9
1.14 Structure of the thesis 9
2 Background 11
2.1 Related work 11
2.2 Fundamentals of Public-key cryptography and PKIs 11
2.2.1 Digital Signature and Verification 12
2.2.2 Hash Function 12
2.2.3 Secure Hash Function (SHA) 13
2.2.4 Certificate Signing Request (CSR) 13
2.2.5 Certificate Authority (CA) 13
2.2.6 X.509 Digital Certificate 13
2.2.7 X.509 Digital Certificate History 13
2.2.8 ASN.1 / DER Encoding 14
2.3 Digital Signature 15
2.4 RSA 15
2.5 Security Assertion Markup Language (SAML) 16
2.6 Java Security 17
2.7 Bouncy castle 17
2.8 Telia's Net iD 17
2.9 HSM 17
2.9.1 General specification and capability 17
2.9.2 Drawback to Using HSMs 18
2.9.3 SafeNet Luna 18
2.9.4 Key Generation performance and key storage capacity 18
2.9.5 Key Export 19

2.9.6 Key storage capacity inside the HSM on the fly and in RAM ... 20
2.9.7 Storage Media ... 20
2.9.8 Timing measurements of the current system as input to the design process ... 20
2.9.9 Maximum FIFO queue length for signing requests ... 20
2.10 Step by step time measurement of traditional HSM's operations ... 21
2.10.1 Key generation time (KG) ... 21
2.10.2 Private Key Wrap and Export ... 21
2.10.3 Private Key import and unwrap ... 22
2.10.4 Signing process time (Si) ... 22
2.10.5 Signing process while performing unwrapping at the same time ... 24
3 Implementation of the proposed solution ... 25
3.1 Database ... 26
3.2 Key Pool ... 28
3.3 Single HSM design ... 29
3.4 Distributed design with more than one HSM ... 30
3.5 Using Physical Keys in distributed systems (using multi HSM) ... 31
3.6 Sign request generator ... 31
3.7 Sign Request handler ... 32
3.8 Signing ... 32
3.9 CA ... 32
4 Analysis ... 33
4.1 Key Generation on the fly ... 34
4.2 Pre-Generated keys ... 35
4.3 Maximum size for the FIFO queue of requests ... 36
4.4 Latency ... 37
4.5 Queue size and Latency Calculation in advance ... 39
4.5.1 Base Rate (BR) ... 39
4.5.2 Queue Size (Q) ... 41
4.5.3 Key Preparation (KP) time ... 41
4.5.4 Latency calculation (L) ... 42
4.6 Reliability / validity Analysis ... 42
5 Conclusions and Future work ... 43
5.1 Conclusions ... 43
5.2 Limitations ... 44
5.3 Future work ... 44
5.4 Reflections ... 44
References ... 45
Appendix A: SAML Signing Request / Response ... 47
Appendix B: Test results ... 51
Appendix C: Luna SA 1700 HSM Performance report (internal by SafeNet) ... 54

List of Figures

Figure 1-1: Signed PDF file can validated by Acrobat reader 4
Figure 1-2: Step by step processing of a digital signature request and response 6
Figure 2-1: SafeNet, Inc.'s Luna SA 1700 HSM as a network appliance..................18
Figure 3-1: Database keys used for storage of data 26
Figure 3-2: Columns of the signature table 27
Figure 3-3: The idle time key generation process and the use of the key pool to process requests 29
Figure 3-4: Single HSM design 30
Figure 3-5: Design with multiple HSMs 31
Figure 4-1: 3000 signing with key generation... 34
Figure 4-2: 3000 signing with key generation on the fly and 2000 additional signing with the one request per second.. 35
Figure 4-3: 3000 sign with pre-generated keys.. 35
Figure 4-4: Latency when the maximum queue size is 50 *without* key pool solution (The vertical lines indicate the arrival rates of the test distribtuion.)... 36
Figure 4-5: Latency when the maximum queue size is 50 with the key pool solution (The vertical lines indicate the arrival rates of the test distribtuion.)... 37
Figure 4-6: Time before first dropped request when using the Key Pool 38
Figure 4-7: Latency versus arrival rate 38
Figure 4-8: The triangle represents the available FIFO queue space and shows how it decreases with increasing arrival rates, at some point being exhausted.. 39
Figure 4-9: Processing of 3000 signing requests at 25 requests per second 40
Figure 4-10: Processing of 3000 signing requests with at 26.3 requests per second 40
Figure 4-11: Processing of 3000 signing requests at 27 requests per second 41

List of Tables

Table 1-1: Gives a summary of the state of NetID plugins 3
Table 1-2: Problems of the current e-ID systems..... 3
Table 2-1: Fields of a X509 Certificate 14
Table 2-2: Number of keys supported by LunaSA SA 1700 18
Table 2-3: Key generation performance of Luna SA 1700 19
Table 2-4: SafeNet key pair generation performance 21
Table 2-5: Time to wrap a private key using AES (256)..... 21
Table 2-6: Time to move a block of RSA 2048-bit keypairs into or outof the HSM 21
Table 2-7: Time to load public key and wrapped private key. unwrap the private key to make the key pair available in the HSM 22
Table 2-8: Performance in Stockholm..... 23
Table 2-9: Performance in Malmö..... 23
Table 2-10: Signing time per key pair with different numbers of simultaneous threads 23
Table 2-11: Signing request time when simultaneously performing a key unwrapping process 24
Table 3-1: Description of the columns of the keys table 27
Table 3-2: Description of the signature related columns..... 28
Table 4-1: Sample of SigningRequest Generation following a specific distribution 33
Table 4-2: Latency calculation vs actual test results 42

List of Acronyms and Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
app	application
ASN	Abstract Syntax Notation
CA	Certificate Authority
CIA	Confidentiality, Integrity, and Availability
CMS	Cryptographic Message Syntax
COTS	Commercial Off-The-Shelf
CRL	Certificate Revocation List
CSR	Certificate Signing Request
DER	Definite Encoding Rules
DSS	Digital Signature Services
EFST	e-Förvaltningsstödjande tjänster (e-Governement support services)
EMR	Electronic Medical Record
FIFO	First In-First Out
FIPS	(U.S.) Federal Information Processing Standard
HSM	Hardware Security Module
IdP	Identity Provider
ITU	International Telecommunication Union
Java SE	Java Platform, Standard Edition
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
KG	Key generation
MAC	Message Authentication Code
NPAPI	Netscape Plugin Application Programming Interface
NIST	(U.S.) National Institute of Standards and Technology
NTLS	Network Trust Link Service
OCSP	Online Certificate Status Protocol
PED	PIN Entry Device
PKI	Public Key Infrastructure
PPAPI	Pepper Plugin API
RSA	Ron Rivest, Adi Shamir, Leonard Adleman
SAMBI	Samverkan för behörighet och identitet inom hälsa, vård och omsorg
SAML	Security Assertion Markup Language
SHA	Secure Hash Algorithm
SITHS	Service identification for both physical and electronic identification.
SITHS	Card has many uses and is suited to all national services in e-health.
SSL	Secure Sockets Layer
SSO	Single Sign On
TSL	Trust service Status Lists
U.S.	United States (of America)

Acknowledgements

I express my gratitude to Professor Maguire who worked tirelessly to see me through the entire degree project process. The company supervisor, Roberth Lundin, also deserves a pat on their back for his counsel and guidance during the design period. Finally, my family has been with me from the start to the end. There is no other way to say thank you, but I am sincerely grateful.

May God bless you all.

1 Introduction

This thesis project designed and evaluated a **key pool** solution for Hardware Security Module (HSM) devices in order to increase their performance by decreasing the response time when processing signing requests in a Digital Signature Service. This chapter provides an overview of the thesis project, describes the research problem in more detail, and specifies the research methodology utilized to carry out this thesis project.

1.1 Problem definition

Today's electronic identification (e-ID) system does not meet the current requirements for e-IDs, hence it needs to be upgraded – especially in terms of advanced embedded security controls. High risk areas include the fact that the authority's access to logs of e-service systems is inadequate. This proposal is supported by the framework agreement established in Sweden for Electronic Identification 2008 (e-ID 2008) that is valid until 30 June 2016 [1]. This agreement calls for the identification of users and these requirements created issues for the transformation to new issuers of e-legitimations. Furthermore, the existing e-ID signature plugin is incompatible with popular web browsers, such as Google's Chrome, Mozilla's Firefox, Microsoft's Internet Explorer, etc.

1.2 New e-ID System

A study of the e-ID system was started by the Swedish government on 17th June 2010 and the complete report of this research was published on December 2010. The report identified a solution for which an Agency under the Ministry of Enterprise was established starting as of 1 January 2011[2]. The acquisition of operations, management of metadata records of all members, guide service, and the other associated tasks were delegated to a new e-ID board (in Swedish "E-legitimationsnämnden"*).

The federation associated with a Swedish Federation of e-identification providers was initiated with its first phase in 2013. The request for quotations ended with only a single quote (from Cybercom Sweden AB), hence this firm eventually got the contract. The definition of a centralized signature service was initiated in 2014. However, this service was excluded from the scope of work and in 2010 was assigned to The Legal, Financial and Administrative Services Agency (Swedish: Kammarkollegiet) blanket e-government services. The framework incorporates six service providers who offered to construct signature services. The approval of these signature services must pass a practical examination process governed by the e-ID board. Moreover, there are other clauses in the agreement that governs the association of Swedish e-identification service federation along with hands on tests conducted during the months of May and June 2014. As per the new clauses of the eID registry board, the authority to purchasd eID is restricted and only the e-ID board is authorized to make such purchases. In March 2014, the Swedish e-ID Federation was formed and started resolving e-ID issues and providing e-services to clients.

The main services were started in "Kammarkollegiets blanket E-förvaltningsstödjande tjänster" (EFST) 2010[3]. Kammarkollegiet invited suppliers, who are part of EFST 2010 to start a new digital signature service based on standalone Security Assertion Markup Language (SAML) and Identity Providers (IdP). During May and June 2014, the eID board started to test and validate the services with regard to all the defined requirements. The first two digital signature suppliers who fulfilled all the requirements were added to the framework agreement.

Initially, the Swedish Tax Office (Skatteverket) directed the very first contract related to Signature Services. The assessments of these contracts were to be announced in September or October of 2014. At the beginning of November 2014, the Tax Office chose Cybercom to supply this service and a contract has been signed.

* http://www.elegnamnden.se

1.3 Signature Service requirements by large Authorities

The E-ID Board has defined some security level requirements for a signature service which must be fulfilled by all Signature Service providers. The performance requirements which must be fulfilled by the service providers include the maximum response time for each sign request during normal hours which is 300 ms and 3000 ms during peak hours.

Additionally, each authority has its own requirements which must be fulfilled by the service providers, such as the maximum request rate at which the provider must be able to respond within the acceptable response time. This information is mostly confidential information and hence not public, but we know that several millions of request per year are expected to be received by these large authorities. Different request rates occur during different hours, days, and months per year. There are several reasons for this. For example, when signing a Tax declaration report and sending it to Tax Office users usually using signing service. As this signing of declarations mostly occurs during the normal working hours of a day and because the traffic is usually high during the months that the Tax Office accepts these declarations (especially in the final hours before the filing deadline) we can estimate the peak request rate. Therefore, we can have made some assumptions about the distribution of request and the request rates during peak hours. In this research, we assume a peak request rate of 50 requests per second[*].

1.4 Plugin problem

Due to its wide use in Sweden, the NetID signatures plugin has been used for this thesis project. Many users over the years have downloaded the NetID plugin because it was easy to use and hassle free. Today it is still very easy to download a signature plugin by clicking on an e-ID application (app), but there are increasing numbers of problems associated with using this plugin.

Microsoft's Internet Explorer customers use an Active-X element NetID plugin. Unfortunately, the NetID plugin's apps are not supported by Windows 8 in "Metro"† mode with Internet Explorer 10. Moreover, it is unlikely that the Microsoft will provide support for this plugin in the future.

For use with Mozilla's Firefox and Google's Chrome, the NetID plugin followed the Netscape Plugin Application Programming Interface (NPAPI). The NetID plugin is supported by Firefox Version 30. Unfortunately, Chrome version 37 will not support NPAPI. However, the NetID plugin still works for the Chrome 35 developer version - because Google's Pepper Plugin API (PPAPI) is still supported. As a result, NetID plugins may not be supported by Chrome during 2015 [6][7].

Today, using the NetID plugin seems to be increasingly awkward to use due to the inability to use Chrome and because users have to answer a number of questions when they use the plugin which is by default blocked by browser. Moreover, many browsers only allow the plugin to run smoothly if it is downloaded from the app store associated with the device/browser vendor. As a result, NetID customers must be charged less (by the app store) and they have to download the plugin for each of the browsers that they use. Telia has realized that this plugin has problems and have announced another method of doing digital signature without using the plugin.

Even if a user has successfully downloaded and installed the plugin, a number of problems can still occur when using this plugin. For example, a number of banking sectors have completely removed the plugin applications from their system. Since March 2014, a Bank ID can be utilized without utilizing any assistance from Active-X elements or plugins. However, all of the user using the old versions of plugins are now enforced to work to use a new (free) app. For Bank IDs this app permits a consistent interface. Unfortunately, because each of the different markets has chosen a different approach to using e-IDs, the end users face a lot of confusion and hassles. This can be seen in the fact that although

[*] This rate for the two busiest hours of the day, half this rate for the remaining business hours, and a quarter of this rate for the remaining hours of the day repeated for a week would be sufficient for signing over 6 million Tax declaraions.

† Metro style was the old name. Now it called Windows 8 Application or windows 8 mode. Internet Explorer app from the home page of the windows 8 is a simplified version of the browser which does not have the same support as the desktop version of Internet Explorer so that is why some additions and features you cannot use it.[4][5]

Telia has distributed a large number of electronic IDs[*], there have not been any endorsements by their customers regarding the use of NetID.

Table 1-1: Gives a summary of the state of NetID plugins

supplier	Version	Comment
Internet Explorer	8-11	Plugins will work as usual until further notice
Firefox	30	Warnings dialog
Chrome	38	Works (but the support of it will be removed in the future)
Opera	22	Works
Safari	7.0	Works

1.5 Purpose

This section discuss several issues related to handling signatures in the ongoing the Swedish e-Identification Board's Certificate Service project for the year 2014-2015. Today browsers are **not** allowed to use signature plugins. This change means that new applications are needed to utilize the emerging new federated e-ID system. These applications can be utilized in order to support any other application as a national service, including applications involving Electronic Medical Records (EMRs). Application that were previously integrated with the earlier browser plugins are now encouraged to run *without* using a browser plugin (i.e., it should run as a stand-alone application).

The problems of the current e-ID systems are summarized in Table 1-2. Section 1.6 will discuss the advantages of this new model for signing, 1.7 describes how a signature can be associated with a specific person, and Section 1.8 describez the design of this new signing service. The bottleneck introduced by the HSM is described in Section 1.9 in order to motivate the goal of this thesis project (as presented in Section 1.10).

Table 1-2: Problems of the current e-ID systems

Asymmetry in the handling of signature	The current e-ID system has an asymmetry in its construction. The design was based upon a report submitted to an authority. In this design a document and its signature are separate files and the handling of each of them is quite different. For example, the signature should be handled as a binary file. This design decision creates huge problems with the management and storage of the document (and its corresponding signature). There is a need for a practical way to create and sign a document. The current Swedish system makes the creation of effective management of the document and their signatures awkward.
Unable to create standard PDF-A documents or XML DigSign files	A signed PDF-A (ISO 19005-1) document **cannot** be created using a Bank ID. In addition, XML DigSig[†] signatures cannot be used in files. XML file results from using a Bank ID need to be handled separately. This is as a major limitation in the use of a Bank ID signature.
Plugins are now seen as the scourge (difficulty) of the browser	Currently plugins are considered a problem when using browsers. The state of the art suggests that plugins and browsers work together, but this requires use of particular methods when using plugins. Each browser manufacture has their own method of accepting and distributing plugins. For example, NetID needs to handle each web browser separately and they need to write documentation about how to install the digital signature plugin for each type of browser.

[*] For example, they distribute e-IDs for the Swedish Tax Office (Skatteverket).
[†] "DigSig" is a project from ebIX which is focused on use of encryption and digital signatures within the European energy sector.[8]

1.6 Advantages of the new service model

A Signing Service requires minimal infrastructure. In the year 2010, under EFST only a SAML ticket was required for a signature or identity data. This SAML ticket can be collected from any identity provider (IdP). SAML allows all the sources of identities, such as smart cards, bank tokens, Bank ID, and mobile cards to activate automatically. In addition, signatures via XML DigSIG, PDF A-2, and XAdES* formats are supported. A digitally signed document is easily created via integration of the signing application with the document generation application, thus enabling a user to produce and store digitally signed documents (as files). The frameworks and assurance are provided due to standard policy of the Federation of Swedish e-ID providers supervised by the e-ID Board. The new Electronic ID system supports "Samverkan för behörighet och identitet inom hälsa, vård och omsorg" (SAMBI) and new federations as well as standalone IdPs. [10]

There is no requirement to keep any of the user's information in the signing server. This reduces the burden on the server and improves security by not storing a user's sensitive information in the server. This will prevent this sensitive information from being stolen from or mishandled by the signing server. A certificate is issued to the user as a time stamped proof of signing. This certificate also specifies the validity time, i.e., the amount of time after signing time that this signature is valid. The certificate can be used for the specified time validity of the signature and the certificate asserts a private or professional identity of the document's signer.

A number of new opportunities are related to the signature services being decoupled by the implementation of the new e-ID system. The main features of the new ID system are that it facilitates a variety of logins and does not require Public Key Infrastructure (PKI) based applications to run on the local system. Moreover, the creation of signatures in other formats is also possible. For instance, a signature can be created for a PDF A-2 document or an XML document. As shown in the figure below, the signature of a PDF-A file can be easily verified using Acrobat Reader.

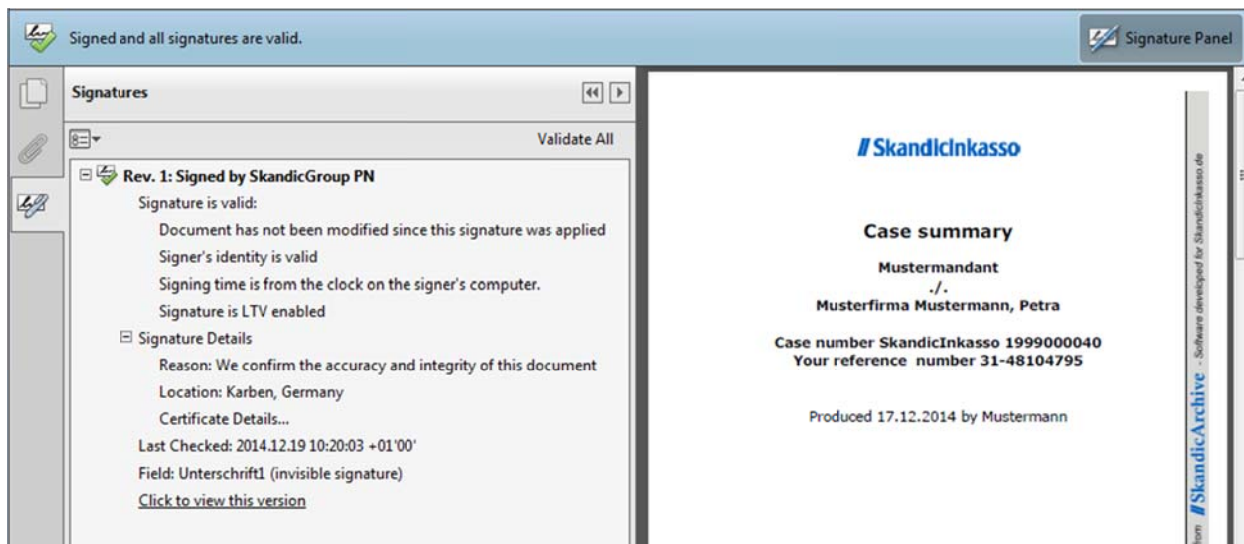


Figure 1-1: Signed PDF file can validated by Acrobat reader

* XAdES stands for "XML Advanced Electronic Signatures" is a set of extensions to XML Signature recommendation making it suitable for advanced electronic signature.[9]

1.7 Proof and validation of a signature in new system

Identity providers can utilize bank issued tokens to enable user to login to the system. When a user sends a document and asks to sign it, the identity provider shows the message to the user and asks the user to confirm the signing request to sign this specific document. After clicking the "confirm" button, the identity provider creates a ticket and places the user information in it. Then the signature service provider uses this identity information to create a certificate for the signed document. Since all of these processes are logged, the identity in the ticket and the identity in the certificate are both identical and linked together. As a result one can used the logged information from the complete process to prove that this signature belongs to that specific bank token and that it belongs to a specific person.

1.8 Service Design

The steps in order for a user to digitally sign a document for an e-government agency are (numbered as shown in Figure 1-2):

1. The user logs in to the authority e-Service (IdP in the background).
2. User asks for sign the document.
3. E-service prepare the request file and send the IdP reference and hash value to signature service provider.
4. The Cybercom Signature Service (CSS) make a call to the identical IdP that the costumer is logged in order to create a "*Proof of identity for Signature (Legitimering för Underskrift)*" to the signature itself. IdP shows a dialog to user and ask to approve that by clicking the button. (Like login process but this time by showing the text that this process is for approve the signature request).
5. Then CSS make a call to signing engine to handle signing the document and certificate creating by a Certificate Authority (CA)
6. Signing engine makes the calls to the HSM, to create the key pair, create the signature, (puts the distinguished name + some certificate extensions + public key in to CSR and make self-signed CSR to CA), destroying the private key.
7. Singing engine send CSR to the CA to create certificate and send back the certificate to signing engine.
8. Signing engine sends back certificate, signing data to the authority e-services.
9. Authority e-services put the certificate, signature in the XML structure or pdf-A document in order to send back to the user.

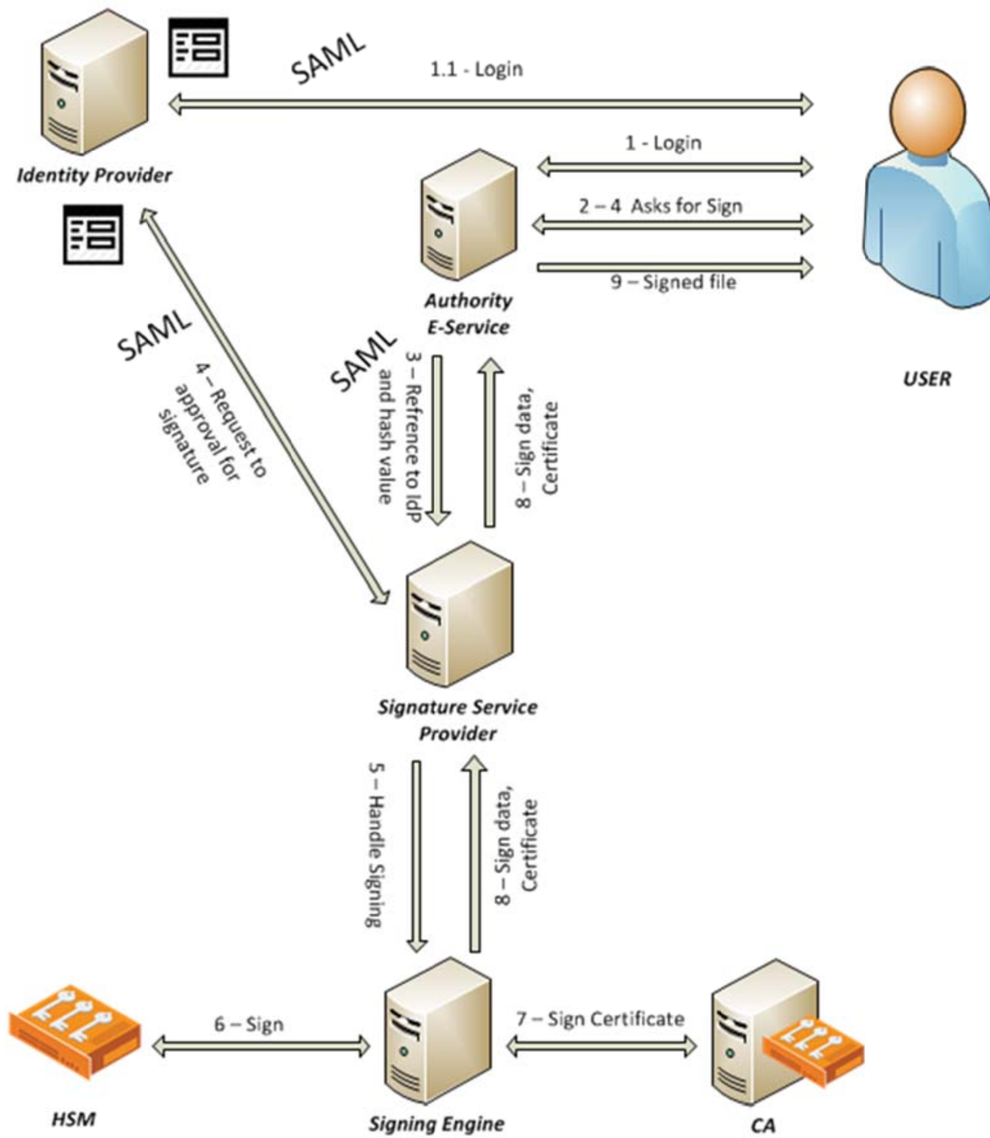


Figure 1-2: Step by step processing of a digital signature request and response

1.9 Bottleneck

From a security point of view availability is one of the edges of Confidentiality, Integrity, and Availability (CIA) Triad. As Matt Bishop has stated, "*Availability refers to the ability to use the information or resource desired. Availability is an important aspect of reliability as well as of system design because an unavailable system is at least as bad as no system at all.*" [11]

This new infrastructure enables companies to provide a general purpose signature service to their customers. Cybercom is one of the companies participating in this competitive market for signature services. It is estimated that some of these signature service providers will receive 200* signature requests per second. In comparison, the current capacity of key generating and signing of SafeNet, Inc.'s Luna SA 1700 HSM, is around 2 signing processes (key generation + signing) per second. The "bottleneck" of the signing process inside the HSM is the key generation process, which requires around 588 milliseconds (see Appendix C). On the other hand, as was discussed in Section 1.3, the signature service provider must be able to handle peak rates of 50 sign requests per second with acceptable latency. One solution could be the purchase of approximately 9† HSMs in order to be able to respond to these 50 requests per second. However, as Section 2.9.2 will describe, this solution would be very costly. Another solution would be to acquire a more expensive but higher performance HSM. Unfortunately, neither of these provides a good solution. Unless this bottleneck is removed the system cannot respond within the required bounded response time and some of the requests will start to be dropped during peak hours, negatively affecting the availability in the signing system.

1.10 Goal

Previous studies have shown that the current HSMs are unable to meet customer expectations because the amount of time that a HSM needs to generate a key pair is too long relative to the expected request rate for signatures during peak hours. The aim of this project is to create a suitable solution by increasing the *effective* performance of an affordable HSM.

In this research project, we propose a solution based upon the introduction of a "Key Pool". The idea, proposed by Prof. Gerald Q. Maguire Jr., is to use the HSM to generate key pairs and store them during idle times. Then during peak hours, these stored key pairs can be utilized in the signing process. This approach of pre-generating keys avoids the need to generate keys at the same rate as the peak arrival rate of signing requests.

However, because the HSM has limited storage capacity, we have to save these pre-generated keys outside the HSM. Moreover, we must be able to store these keys outside of the HSM without compromising the assurance level of the HSM. In order to maintain the desired level of assurance, we use well-trusted encryption methods to protect the pre-generated keys in our "key pool" when they are stored outside of the HSM. The ability to securely store parts of the key pool outside of the HSM decouples the number of keys that can be pre-generated from the memory capacity of the HSM.

The aim of this research is not only to optimize the effective performance of a single HSM in order to meet the expected customer demands while saving money, but also to create an extensible set of HSMs to respond to even higher customer demands in the future. This later is possible if it is possible to load the pre-generated keys into one of several HSMs that can simply be used for signing during periods of peak demand for signing.

---

* This number is only an estimated number. As it discussed in Section 1.3 most of the actual peak rate and distribution of requests information is confidential and not shared by large authorities.

† It depends on the request distribution and duration of peak hours. In this research two hours was selected as the duration of the peak rate of requests.

1.11 Aim of this master thesis

The currently available high performance HSMs are not cost-effective on large scales to reach the performance levels expected in the near future with the new signing method (discussed in Section 1.2). Digitally signing documents is expected to be a large business and socially very important to citizens and businesses, hence it is worth some research in order to go beyond the performance limits of the currently available HSM platforms. There are various approaches that could be used. This thesis seeks to improve the process of key generation using two methods. A detailed analysis of the time required for each stage of the process will be examined from beginning to the end. The aim is to introduce a solution that will make the use of digital IDs both feasible and economical, in order to take advantage of the features that digital IDs have over the traditional methods of signing documents.

1.12 Research Methodology

The thesis applied the design science methodology for this research. A working artifact demonstrating the feasibility and enabling performance measurements of the proposed solution was designed and evaluated. The project took place as the following steps:

- Existing solutions for digital signing services were investigated using a literature study. All of the procedures involved these services were investigated by examining the operation of a HSM in detail (based upon examining a SoftHSM, the SafeNet HSM simulator, and the SafeNet Luna SA network-attached HSM). The literature study also looked at Java security and details of certificates; CAs; PKIs; Ron Rivest, Adi Shamir, and Leonard Adleman (RSA) public-private key pairs; etc.
- The different methods involved in the signing procedure were investigated, measured, and analyzed in order to and discover the limiting step(s) in this processing. This lead us to split the investigation into the following components:

    1. Understand how the signing requests arrive and are place in a first in-first out (FIFO) queue.
    2. Measure the time required for each of the following operations: batch RSA key pair generation by both a soft HSM and actual hardware, wrapping, exporting, and saving the generated keys in two different media (specifically files and a database), loading the wrapped keys from a file or database, and unwrapping the keys.
    3. Measure the time required to sign simulated request bytes[*] when generating the RSA keys for each request on the fly.
    4. Measure the time required for different types and sizes of keys.

- The key pool solution is introduced in order to use pre-generated keys. The likelihood of having a sufficient number of pre-generated keys is examined in order to assess the efficacy of the proposed solution for a high performance signing service. An analysis was done to learn the maximum number of signing requests per second that could be processed over the course of a day (where the actually load varies between very low and much higher than the HSM’s maximum service rate).
- The proposed key pool solution was implemented.
- The performance of the proposed key pool solution was measured using a Luna SA HSM and these measurements were evaluated by comparing the performance of the prototype with the results of existing solutions.

[*] Simulated requests were used to avoid compromising any real customer’s data and to enable the analysis of much higher signing request rates than currently occur in practice.

1.13 Delimitations

The effective performance of a signing service was enhanced in this thesis project. The functions related to the signing service were revised to reduce the maximum delay for a signing request. The proposed solution can overcome the processing rate and storage limitations of the present HSM due to the proposed change in the method of key pair generation.

The main focus of this project concerns the operations performed by the HSM (in terms of limitations of the performance of the signing service). Moreover, the signing service information along with the private information store in the HSM, the private results of the key generation, etc. remain private and confidential – despite the key wrapping/export and import/unwrapping operations. As the details of the HSM are highly confidential to the manufacturers, this thesis proposes a means to improve the performance of the HSM – ***without*** the need to consider the details of the HSM. Therefore, the security factors of the HSM based system are not highlighted in this thesis. In fact, the implementation details of the proposed solution and how to ensure the integrity of the overall system are only briefly mentioned.

Additionally, some elements of the HSM that can affect the performance of the solution, such as the signing algorithm, key size, wrap key type, and the size of the wrapped data are not discussed in detail in this thesis. However, detailed information is provided concerning the CA certificates, the structure of these certificates, and samples of same of the elements present in these certificates.

1.14 Structure of the thesis

Chapter 2 provides background information related to a Public Key Infrastructure. The core elements and the characteristics of such an infrastructure are described briefly. Moreover, a detailed discussion is given about the HSM and its functions as related to the Public Key Infrastructure.

Chapter 3 discusses details of the processes of key generation and signing, and then introduces the proposed solution using a key pool. The chapter also gives some details of the implementation of the proposed solution.

Measurements of the performance of the prototype are analyzed and compared with existing solutions in Chapter 4. The analysis of these measurements influenced the subsequent further development of the prototype. The results of this analysis are an improved prototype that meets the goal specified in Section 1.10.

The final chapter of this thesis presents some conclusions, a discussion of how well we achieved our aims, and what was gained by implementing the proposed new solution. In addition, the effect of the new solution is described in terms of the context of the initial problem. Some future work is proposed to extend the results of this thesis project and some reflections are made regarding the social, economic, and other aspects of this thesis project.

2 Background

A digital signature realizes a scheme to assert information about security components, such as the privacy of a conversation, the integrity of data, the authenticity of a digital message/sender, and non-repudiation by a sender. Digital certificates are facilitated by the existence of a PKI. Digital certificates are the main component of a digital signature service.[12], [13]

A CA signs a certificate to attest to its authenticity. This certificate can be used to create a digital signature. The certificate can be stored in hardware devices or in a file stored on a storage device.[14]

The history of certification methods starts in 1976, when public key cryptography first appeared. Due to the threat of "man-in-the-middle attacks" researchers developed certificate based methods to provide users with confidence in the authenticity of the public keys they are using.

The guarantees developed are based upon digital certificates signed by trusted entities, referred to as CAs. A CA vouches for a particular public key belonging to its indicated owner. The emergence of PKIs led to the deployment of mechanisms to manage digital certificates throughout the existence of the corresponding keys. However, the certificate based infrastructure that developed has suffered because PKIs turned out to be very complex to deploy, cumbersome to use, and non-transparent for users.

Based on key pairs and digital certificates, a PKI facilitates the use of public key cryptography. Today, a lot of Commercial Off-The-Shelf (COTS) software, such as e-mail programs, web browsers, file encryption software, and groupware, has some form of PKI support embedded in it [13]. These PKI enabled applications are the main beneficiary of the results expected from this thesis project.

Another crucial part of a PKI is key management. Key management involves the generation, exchange, storage, use, and replacement of keys [15]. In a service that requires high security standards, such as FIPS 140 level 2, level 3, or level 4 [16], all cryptographic and key management processing must be handled by a specific cryptographic module called a hardware security module (HSM).

2.1 Related work

Research has previously been mainly done on improving the performance of the signing process of HSMs. In contrast this research focuses on ***key generation*** by HSMs, but no previous research was found regarding improving key generation by HSMs.

2.2 Fundamentals of Public-key cryptography and PKIs

Cryptography is the science of encoding and safeguarding data. Public-key cryptography has been in existence for a long period and a large amount of research & development work has been done. Different committees have proposed standards related to public-keys and PKI. In order to understand the rest of this thesis one must understand the details of how public-key cryptography and PKIs actually work. First we look into relevant details so that the reader of this thesis will understand the operation of public-key encryption and digital signatures.

This section describes some of the basic elements of public-key cryptography and PKIs. Understanding these details will help us to explain the software, hardware, and procedures necessary to achieve the level of trust expected by society and industry.[13]

Public-key cryptography is distinguished from a symmetric-key, private-key, and shared secret approach by enabling one key to be made public and the other key is kept private. In contrast, symmetric-key cryptography uses a common key for both decrypting and encrypting messages. This is intuitively similar to the expectation that one can lock and unlock a door using the same key. However, this method requires a secure way to distribute the secret-key to the two (or more) parties.

Public-key cryptography utilizes key pairs, one for encrypting and another for decrypting. A message encrypted using a public-key can only be decrypted using its corresponding private-key; while a message encrypted with the private-key can only be decrypted using its corresponding public-key. This asymmetry is used in the implementation of digital signatures and encryption. This concept is very attractive and it offers a number of advantages as compared to symmetric-key cryptography. One of these advantages is that one party can apply their private key to digitally sign a message, while the validity of this signature can be checked by anyone who has a copy of the signer's public key. Additionally, the use of a public key for this operation simplifies key distribution – as each user's public key can be published widely without compromising the user's private key (provided that it is *very* difficult* to use the public key to find the corresponding private key). Only the user's private key needs to be carefully protected.

2.2.1 Digital Signature and Verification

Creating a digital signature provides a means by which a message can be verified or authenticated, i.e. proving that the message originates from a specific sender. For example, if Bill wants to digitally sign a message and send the result to Tom. Bill uses his private-key to encrypt the message; he then sends this encrypted message along with his public-key (the public key is attached to the signed message) to Tom. Tom applies Bill's public key to see original message. However, at this point Tom has no way to neither know that it is actually Bill's public key nor know if the message has been tampered with.

The possibility of combining the digital signature and the encryption enables the communicating parties to have both privacy and authentication. Encryption using the recipient's public key can be used to ensure the privacy of the message, since only the recipient has the corresponding private key to decrypt the message. While as described above we can use a digital signature to enable the recipient to authenticate the message.

Unfortunately, the time required to perform public-key encryption is typically much greater than the time required to do encryption using a symmetric-key. In contrast, distribution of public keys for use with public-key cryptography is simple (as no secrecy is required), while secure distribution of symmetric keys is difficult. This leads to three interesting results: (1) we can use public-key cryptography to help us distribute the symmetric key, then use the symmetric key for decrypting a potentially large message, (2) we can compute a hash over a large message to produce a short (even fixed size) hash and then securely transmit the hash to the other party – who can now easily check if the message it authentic and if it has been modified, and (3) we can use hashing together with digital signatures to provide authentication of origin, detects any modification of the message, and achieve non-repudiation.[17, Ch. 6]

2.2.2 Hash Function

Any function that maps data from arbitrary length datum to a fixed length datum is referred to a hash function. The output of the hash function is called a hash value, hash code, hash sum, or checksum depending upon how this output is used. Hash functions are used to generate fixed length output that acts as a reference to the original data. This is handy when the original data is very cumbersome to be used in its entirety. Hashing can be thought of as a one-way encryption algorithm. We say that it is one-way, because it should not be feasible to derive the original message from the hash.

A practical illustration of the application of hashes is the data structure known as a hash table. In this structure, data is stored in an associative manner. Using a hash table minimizes the search time in comparison to a linear search as the hashed value is used to locate the table entry for a potentially long string.

---

* This is usually evaluated relative to an advisory's assumed available computing power, such that the advisory will not be able to find the private key in less time than the message should remain unencrypted, unforgable, etc.

Generating hash values from the input data facilitates verification that the data matches the expected data. Because generating a specific hash value is not easy and it should be unlikely that two different strings generate the same hash value, hashing has been widely used to verify that the received data has not been modified since it was sent.[18, Ch. 5]

2.2.3 Secure Hash Function (SHA)

The United States (U.S.) National Institute of Standards and Technology (NIST) published a Secure Hash Algorithm (SHA) as a U.S. Federal Information Processing Standard (FIPS): SHA-1 is one of the members of a family of cryptographic hash functions. NIST announced that SHA-1 is in end of life. SHA-2 is another family of hash functions, with different block sizes. SHA-2 includes SHA-256 and SHA-512. These algorithms differ in the word size used in the algorithm; SHA-256 uses 32-bit words while SHA-512 uses 64-bit words. There are also truncated versions of each standard, known as SHA-224 and SHA-384.[18, Ch. 5]

2.2.4 Certificate Signing Request (CSR)

A CSR is an encrypted text in a file that is derived from the computer server from certificate owner information like organization name, common name (domain name), locality, country, email address and distinguished name and the public key that will be included in the certificate. Then this file will be sent to CA to perform the sign and create the certificate.

2.2.5 Certificate Authority (CA)

A CA is an entity that issues digital certificates. A CA certifies that the named subject of the digital certificate is the owner of the public key contained in this certificate. (The structure of a digital certificate is describe in the next section) This certification enables third parties to rely upon signatures made using the private key that corresponds to the public key that was certified. In a typical trust model, a given CA is a trusted third party who is trusted by all the communicating parties. Most public key infrastructure schemes feature one or more CAs.

Note that it is also possible to create a hierarchy of CAs with each CA's certificate signed by the CA above them in the hierarchy. It is also possible for two CA hierarchies to cross certify their certificates. For example, two organizations might agree that certificates signed by either hierarchy of CAs would be trusted by applications used in these two organizations.[18, Ch. 7]

2.2.6 X.509 Digital Certificate

In 1988, the International Telecommunication Union (ITU) established a certificate standard called X.509 in order to enable authentication of remote network users. X.509 is based on public key cryptography and data signatures. In X.509 the digital certificate contains [19]:

- A version number which indicates the version of X.509 that the certificate's data format follows.
- A public-key certificate is a digitally signed statement from one entity, indicating that the public key of another entity has some specific value.

Further details of the X.509 certificate's structure are given in the following section.

2.2.7 X.509 Digital Certificate History

X.509 Version 1 is the most generic and the most widely used version of X.509. X.509 Version 2 introduced the use of unique identifiers for both the subject and the issuer to enhance the possibility of reusing the subject, the issuer, or both.

In 1996, X.509 Version 3 was developed to support extensions. These extensions allow anyone to define an extension and include this extension in a certificate. There are number of extensions that are used, these include:

- Key usage to limit the use of keys to a particular purposes such as “signing-only”
- Alternative Names associates other identities with a given public key, examples include: DNS names, Email addresses, and IP addresses.

In order to indicate that an extension needs to be checked, the extension is marked as being critical and set to "*keyCertSign*". For example, if such a certificate is presented during Secure Sockets Layer (SSL) communication, the receiving system should reject it as the extension indicates that the associated private key is meant only for signing certificates and hence should not be used in SSL[19].

Table 2-1: Fields of a X509 Certificate

Version:	The Version field identifies the version of the X.509 standard utilized in this certificate; this affects the information specified in the certificate.
Serial Number	To differentiate one certificate from the other, the entity creating the certificate assigns a serial number to it. This information is of great importance. For example, when a certificate is revoked the serial number of the certificate is placed in a certificate revocation list (CRL).
Signature Algorithm Identifier	This field shows which algorithm used by the CA to sign the certificate.
Issuer Name	The Issuer Name is the name of the entity that signed the certificate and this is normally a CA. Use of this certificate implies trust of the issuer of this certificate. Note: In some cases an issuer signs its own certificate (called a self-signed certificate).
Validity Period	Every certificate is only valid for a stated period, after which it becomes obsolete. This is the certificate’s validity period. This period is usually specified as a start date and an end date. The validity period chosen for a given certificate is dependent on factors such as how strong the private key is and the amount a user is willing to pay to acquire the certificate.
Subject Name	Subject Name refers to the name of the entity identified by the certificate’s public key. The name expected to be unique across the internet. For example, a Distinguished Name (DN) of an entity might be: • Common Name (CN)= Test Name, • Organizational Unit (OU)=IT Co, • Organization (O)=Cybercom. • Country (C)=SE
Subject Public Key Information	Subject Public Key Information is normally used to refer to the public key of the entity that is being named in conjunction with the algorithm identifier. This identifier must specify the cryptographic system to which the public key belongs and the parameters that are associated with it.
Extensions	Allow anyone to define an extension and include this extension in a certificate.[19].

2.2.8 ASN.1 / DER Encoding

Data contained in a certificate is encoded using Abstract Syntax Notation (ASN) 1/ Definite Encoding Rules (DER) standards. DER describes a single way in which data can be stored and transferred.[20]

2.3 Digital Signature

A digital signature is a mathematical scheme used to provide a number of assurances, such as the privacy of a conversation, integrity of data, the authenticity of a digital message or sender, and non-repudiation of the sender. In cases where one may be concerned about the security of sensitive documents such as receipts, contracts, agreements, or other similar documents where users are concerned over unauthorized access or theft of data, the best solution is application of a digital signature.

When a document is digitally signed, the signature is usually sent as a separate document. A recipient of a digitally signed document receives both the message and the signature and he/she applies a verification technique to the combined message and signature in order to verify the authenticity of the digital signature of this document. Digital signatures prevent unauthorized changes to a document. Additionally, successful verification of a digital signature ensures that the expected party has signed the document that has been received. Encryption can be used to ensure the privacy of the message.

Digital signatures are used to in conjunction with efforts to ensure privacy, authentication, integrity, and non-repudiation. As an authentication mechanism, digital signatures enable the message sender to attach a unique code that acts as a signature. This signature is normally formed by computing a hash over the message and encrypting the resulting hash value with the sender's private key. The advantage of this technique is that the signature gives a guarantee of the source and the integrity of the message. Following the NIST standard, a digital signature uses a secure hash algorithm (such as SHA-512) to compute a secure hash over the plain text message. Next the plaintext message, the message signature, and the public key of the sender are packed together, signed, and encrypted using the public key of the recipient. The recipient unpacks the received message, decrypts the message using its own private key, then the same hash function is used to compute a message digest of the received message which is compared to the decrypted signature. If the message digest and the signature match, then the message is believed to be authenticated and unmodified.[14]

2.4 RSA

In the 1970s, Ron Rivest, Adi Shamir, and Leonard Adleman invented an encryption algorithm that has been named after them. RSA relies on the difficulty of factoring integers. This scheme is the best known and most widely used public key encryption scheme. It is based on the concept of exponentiation in a system that shows some degree of congruence over the integer's module (a product of two primes). It usually makes use of large integers (for instance 1024 bits) and its security is based on the assumption that factoring large numbers is difficult, i.e., that factoring is computationally expensive. For example, factoring a value "n" using a standard factoring algorithm takes $O(e^{\log n \log \log n})$ operations. Therefore, in key generation, each person creates their own private and public key pair as in ElGamal. [17, Ch. 4]

RSA Key Generation of a public/private key pair generation consists of the following steps:

- "Select two random large primes, p and q;
- Compute the system modulus $n = p * q$; and
- Encryption key e could be a random selection, where $1 < e < \phi(n)$, $\gcd(e, \phi(n)) = 1$ (note $\phi(n) = (p-1)(q-1)$);
- Now we can calculate the decryption key d: $e * d = 1 \pmod{\phi(n)}$ and $0 \leq d \leq n$.

Public encryption key: PU = $\{e, n\}$
Secret decryption key: PR = $\{d, p, q\}$."[17, Ch. 4]

Assume that we have two parties: Bill and Tom. Tom wants to use this scheme. Bill sends a message to Tom. This message must be in the form of a number that is acceptable by Tom's modulus system. This problem can be implemented in the same way as the ElGamal scheme, thus the message m has to be smaller than the modulus n. Bill can break his message it into blocks if necessary.[17, Ch. 4]

2.5 Security Assertion Markup Language (SAML)

Two previous security initiatives are the source of the Security Assertion Markup Language (SAML): the Authorization Markup Language (AuthXML) and Security Services Markup Language (S2Ml). Entities which have identity related information that is specific to a given security domain is termed a subject. The framework for the exchange of the security information in SAML is an XML-based system on subjects. It does not require the use of a single vendor's security architecture, as SAML does ***not*** provide the underlying user authentication design.

SAML is made up of components that when combined enable various functionalities to be implemented. These components can be used to provide a transfer of authentication or transfer attribute and authorized information among autonomous firms that have created a trust relationship. SAML defines the content and structure of protocol messages that are used for both the transfer of the information and assertions. In the latter case, an assertion in SAML is written using XML. Additionally, SAML profiles have been defined to meet the needs of particular business functionality, for example the Single Sign On (SSO) profile.

The standard-based approach provided by the SAML enables SSO among numerous applications and supports identity management. Prior to the introduction of the SAML standard, enforcing security by developers focused on using proprietary security mechanisms leading to heterogeneous application systems. The result of using these proprietary security mechanisms was cost-ineffective and lead to interoperability issues between different vendors products. In many cases, this required the system to use client side screen scrapping or in the worst-case scenario using key stroke logging. The lack of interoperability and the ***ad hoc*** solutions to overcome these problems introduced many security loopholes and increased the risks of client side hackers. Moreover, the resulting patchwork solutions made it difficult to manage deployment and troubleshooting of the integration of multiple applications. The design of SSO enabled the user to sign on to different application hence solving the problem.

A proprietary mechanism can be used to encrypt the user credentials in the HTTP-POST header and to pass the security token to another application, thus encapsulating the details of the proprietary mechanism. The security of this transfer can be realized through a secure transport protocol such as SSL. When the user authenticates using the SSO-enabled application, the client application uses the SSO mode of the security token that is available in the HTTP-POST header. This design helps to redirect the user to the target resources or the application which has the appropriate access privileges. Proprietary agents interpret the HTTP header that contains the SSO security token. The use of a particular proprietary agent is common to a business and their trading partners. A vendor-defined mechanism can be similar to this approach, yet follow the fundamental SAML specification for the representation of authentication and authorization of the credentials for the standard security-token format.[21],[22]

2.6 Java Security

A set of the application programming interfaces (APIs) is included in the Java security technology, tools, and the execution of common security algorithms, mechanisms, and the protocols. The area spanned by the Java security protocols is wide and includes cryptography, public key infrastructure, secure communication, authentication, and access control. A comprehensive security framework provides this technology. Moreover, Java security provides the system administrator with a set of tools which aid him/her in ensuring that the application secure. The Java security platform offers dynamism, extensibility, standardization, and interoperability.[23] The features relevant to this thesis project are cryptography, authenticity verification capability, public key infrastructure, and authorization.

2.7 Bouncy castle

Bouncy Castle Crypto is a package used in the Java implementation of cryptographic algorithms. It was developed by the Legion of the Bouncy Castle. This package provides a lightweight API that is suitable for use in any environment that conforms to the Java Cryptography Extension (JCE) framework. Bouncy Castle can generate X.509 certificate in both version 1 and 3 which are used in the CA implementation. [24]

2.8 Telia's Net iD

Telia's Net iD utilizes a smart card and PKI to support e-identification in Sweden. The product has two groups of potential customers: (1) health care, municipalities, and government agencies and (2) Telia's customers. Telia no longer provides smart cards containing certificates to private customers, instead individuals can get a smart card ID from the Swedish Tax Office (Skatteverket).

2.9 HSM

The section gives a general description of HSM devices, including the specific functionalities that were employed in the design and evaluation of the proposed solution. This description will be important to understand the tests described in Chapter 3.

Different vendors' HSMs have different functionalities, in addition to the basic functions of key generation and key storage. Additionally, the different vendors have different shares of the market. Key export is a function that can be found in some HSMs, but it may only be available in specific models of a vendor's products. In this thesis project, we utilized a SafeNet Luna SA 1700. All of the measurements were made using this device and the functions that will be described in the following subsections describe the features of this specific HSM.

2.9.1 General specification and capability

A HSM is a cryptographic processor that is specifically designed to be used for the protection of a cryptographic key throughout its lifecycle. HSMs act as trusted anchors to protect a PKI. They are designed to utilize cryptography in order to protect some of the most security conscious organizations in the world. This protection is achieved by managing, processing, and a storing cryptographic keys securely inside a hardened and tamper resistant device.[25]

A HSM is capable of performing a number of important security related functions, including:

- Cryptographic operations, such as encryption, digital signatures, hashing, and computing Message Authentication Codes (MACs).
- Key management functions such as key generation and secure key storage.
- Authentication by verifying digital signatures.[26]

2.9.2 Drawback to Using HSMs

The major drawback to the use of a HSM is its cost. The price of these devices can range from under a thousand U. S. dollars to many thousands of dollars. Their cost depends on the level of functionality and the sophistication of the security required by the customer. In addition, a HSM requires support and maintenance adding to the cost incurred by customers that have purchased a HSM.[26] A network appliance version of the SafeNet Luna SA 1700 that can do key exporting costs ~US$25K [27]. One of the reasons for the high price for these devices is the high costs for HSM vendors getting a certificate (from a testing laboratory) that their product meets government specifications such as NIST FIPS 104-2.

2.9.3 SafeNet Luna

SafeNet, Inc.'s Luna SA 1700 HSM was designed with the security of its cryptographic elements in mind. This HSM is a popular choice for enterprises requiring a secure mechanism for storing cryptographic keys that is both strong and trusted. The SafeNet Luna SA 1700 product was designed to address market needs where security is very important. SafeNet Luna can be easily integrated in a number of applications to accelerate cryptographic operations, to secure the cryptographic keys for their full lifecycle, and to act as a root for an enterprise's entire encryption infrastructure.[28]

Figure 2-1: SafeNet, Inc.'s Luna SA 1700 HSM as a network appliance

2.9.4 Key Generation performance and key storage capacity

Table 2-2 shows the memory capacity of the HSM for various types of keys for two different memory options: 2 Megabytes of base memory and 16 Megabytes with a memory upgrade. Table 2-3 shows the performance of a Luna SA 1700 with either one or two HSMs for key generation, hashing and signing, and signing. All of the timings in this thesis are given in units of milliseconds (ms). Additional information about this device is given in Appendix C.

Table 2-2: Number of keys supported by LunaSA SA 1700

Key type	Base memory	Memory Upgrade
AES 256-bit	16,000	129,000
RSA 1024-bit	4,500	35,000
RSA 2048-bit	2,500	20,000
RSA 4096-bit	1,200	10,000

Table 2-3: Key generation performance of Luna SA 1700[*]

		Single HSM		2xHSM	
Operation	**Key size (bits)**	**Ops/second**	**Latency (ms)**	**Ops/second**	**Latency (ms)**
KeyGen	1024	11	98	3.6	210
	2048	1.8	590	1.3	650
	4096	0.17	15,000	0.067	10000
	8192	0.008	120,000	0.025	40000

2.9.5 Key Export

SafeNet Luna HSMs originally performed all cryptographic operations within the HSM and only allowed the ***results of these operations*** to be available outside the HSM. Fortunately, the current HSM provides the functionality necessary for backup and restore operations. Based upon this functionality we devised a secure hardware mediated transaction that implements cloning[†]. This allows sensitive materials to be moved directly between HSMs in a secure fashion. This technology was limited to handling very large numbers of keys, i.e., more keys than the storage capability of the HSM.

Some developers and service providers have made it possible to store key materials outside of the HSM, for example in databases or in other suitable frameworks. These solutions permit large numbers of keys to be stored and facilitate management of security information.

The solution that SafeNet Luna offers comes in two different versions to addresses different application requirements, specifically:

1. For those applications that demand optimum physical and procedural security the export of any material that is deemed to be sensitive is not permitted.
2. For applications that make use of databases that contain key and profile materials, there is a special version of the HSM that permits export of materials in encrypted form for storage in an external database. These materials can later be imported, decrypted, and used within the HSM. Note that these materials *cannot* be used outside the HSM, only stored and returned to the HSM.

The two versions of the HSM are mutually exclusive. However, the non-exporting version is capable of storing and handling sensitive objects that are contained in the HSM and these objects can be copied and moved directly to another similar HSM through the cloning operation. It is also important to note that the exporting version is capable of wrapping key materials exported from the HSM and unwrapping key materials imported into the HSM, but this version of the HSM is not capable of performing cloning. Moreover, it is not possible to convert one version of the HSM into the other version without destroying all the contents of the HSM. Therefore, the exporting model of Luna HSM provides a key export capability while running in full FIPS 140-2 level 3 validated mode of operation. The advantage of this model is that you do not need to downgrade the security of the HSM device in order for the export capabilities to function.

The exportability of the objects (keys) is an important consideration as the keys must only be accessible and used inside the HSM in order maintain the trust level of the system. When the HSM's

[*] SafeNet never publishes their product performance. The numbers used in this document are based on the latest test results from a SafeNet Sales engineer. The full set is given in Appendix C.

[†] In this context cloning means taking a full back up from a partition of the HSM and storing that in a backup device called "Luna Backup HSM". The resulting backup partition can be restored into other HSM. This feature can be used in Key Pool Distributed mode. See Section 3.4.

export capabilities are disabled, keys can never leave the HSM. In cases where the organization requires the two different capabilities, the use of a mixed population of HSMs is encouraged. This involves the use of different Luna HSM servers to address each requirement. Setting of the Luna HSM configuration is done at the factory with the provision that setting the system requires contacting the vendor and shipping the HSM back to the company for reconfiguration.[29]

2.9.6 Key storage capacity inside the HSM on the fly and in RAM

After decoding the keys, testing was performed in order to retrieve data into the HSM's random access memory (RAM). The maximum number of the RSA 2048-bit keys pairs which can be stored in the RAM (on the fly) of the HSM was determined to be 12,033. Note that the figure here differs from the value in Table 2-2 since the table illustrates HSM memory values in numbers of keys, rather than numbers of key pairs. Alternatively, 12,033 key pairs represent the number of key pairs that can be unwrapped and stored in the RAM.

2.9.7 Storage Media

Two different types of storage were utilized for testing. First, a flat file was used for storing the public and private keys that were wrapped and exported. The second type of storage utilizes a database. Using a database provides greater convenience and flexibility when storing the exported data, while also permitting other metadata to be stored along with the exported wrapped sets of keys. Sections 2.10.2 and 2.10.3 described the results of the measurement using these two different types of storage.

2.9.8 Timing measurements of the current system as input to the design process

Before we initiated the design and implementation stage of our proposed solution, the critical processes associated with key generation, hashing & signing, and signing were measured. For each process, a test was executed many times in order to provide reliable data as input to the design stage.

2.9.9 Maximum FIFO queue length for signing requests

A set of measurements on Luna SA 1700 were made to determine the maximum length of the FIFO queue. These measurements were based on injecting a very large number of requests at an increasing rate and watching when the number of responses was no longer in equilibrium with the injected rate, i.e., when requests were being dropped. The maximum queue length for requests was found to be 150 requests.

As 150 is the maximum queue size, this means that the next request in the queue (which would have the number 151) would have a maximum total response time of 3020 ms (3000 ms waiting + 20 ms sign) – which exceeds the maximum acceptable response time defined by the eID board (e-legitimationsnämnden). From this we can conclude that the 150 requests have to be processed within 3000 ms, which is equivalent to 50 requests per second.

This small queue size is potentially a system bottleneck as the system will start to drop requests which could have been processed successfully within the acceptable maximum response time of the burst length of requests exceeds 150 requests in 3000 ms (assuming a processing rate of 50 requests per second).

In the Analysis in Section 4.3, in order to be able to perform tests of both the traditional and new key pool systems, the maximum queue length has been set to 50. The reason for using 50 rather than 150 is that 150 is quite a large number for our test scenario and it is not possible to fill this queue and get dropped requests within the 3000 signing requests that we have used for our test request distribution (as specified in Table 4-1).

2.10 Step by step time measurement of traditional HSM's operations

This section concerns testing and measurement of operations that are essential. These results were used as input to the design and implementation described in Chapter 3. These values will also be utilized for comparison with the performance of the prototype of the proposed solution.

2.10.1 Key generation time (KG)

The new system requires RSA 2048-bit key pairs (by default) for testing in the test environment. The key generation rate was on an average ~673ms per key pair. This value was close to the SafeNet product specification of the performance of key generation for RSA 2048-bit (compare the results in Table 2-4 with Table 2-3).

Table 2-4: SafeNet key pair generation performance

Number of key pairs	Total time (ms)	Time per keypair (ms)
100	66,462	664.62
1,000	683,499	683.499
2,000	1338,983	669.491

2.10.2 Private Key Wrap and Export

Measurements of the time to wrap and export keys stored in RAM to external storage (to a file and to a database) were conducted when using AES 256 to encrypt the block of keys. Table 2-5 shows the time to wrap and export a block of RSA 2048-bit key pairs to a file, while

Table 2-6 shows the time to load RSA 2048-bit key pairs from a database. The time to transfer 10,000 wrapped RSA 2048-bit key pairs to and from the HSM and the database is less than 1 ms/key.

Table 2-5: Time to wrap a private key using AES (256)

Number of keys	Total time (ms)	Key pair generation, wrap, and save to file (ms)	Key generation per key (ms)	Wrap and save to file per key (ms)
100	85,012	850.12	709.25	140.87
200	167,575	837.875	701.715	136.16
1,000	810,143	810.143	688.92	121.223

Table 2-6: Time to move a block of RSA 2048-bit keypairs into or outof the HSM

Number of key pairs loaded from DB	Total time (ms)	Time per keypair (ms)
100	485	4.85
500	635	3.27
1,000	577	0.577
5,000	829	0.165
10,000	953	0.095

Number of key pairs writen to DB	Total time (ms)	Time per keypair (ms)
100	560	5.6
500	1,543	3.08
1,000	1,507	1.5
5,000	5,168	1.03
10,000	9,437	0.94

2.10.3 Private Key import and unwrap

Table 2-7 shows the results of the entire test of importing key pairs and un-wrapping the private key. It takes on average 70 ms to process each private key.

Table 2-7: Time to load public key and wrapped private key. unwrap the private key to make the key pair available in the HSM

Number of key pairs	Load from DB (ms)	Cipher initialization (ms)	Unwrap private key process (ms)	Total time (ms)	Unwrap time per private key (ms)	Time per key pair (ms)
1,000	537	2,719	69,958	74,525	69.958	74.525
2,000	701	2,867	140,143	144,245	70.0715	72.125
3,000	693	2,721	209,628	214,225	69.876	71.408
4,000	778	2,718	279,751	283,433	69.937	70.858

2.10.4 Signing process time (Si)

The signing process is considered to be an essential security function that needs additional attention. As per the specification of the HSM, the device is capable of processing 1200 signing requests per seconds, hence less than one millisecond for signing per multithreaded process.

The tests conducted in this thesis project utilized two test environments located in two different geographical locations, i.e. two different cities. The first test environment is located in Malmö and the other test environment is in located Stockholm, both in Sweden. The time taken for sending packets between these two sites is ~10 ms. In per our testing procedures, we perform two signing process for every signing request, i.e. when signing a document with a new key pair and signing certificate with CA. If a test is initiated from the Stockholm site, the delay to send a request to Malmö and receive a response will be ~20ms, which is more than the time required to perform the signing request. All of our tests were initiated at Stockholm site.

Table 2-8 and Table 2-9 show the signing time per key pair at the two different times (excluding the network delay). It should be noted that the processing time per key pair decreases with increasing numbers of requests. There are two main reasons for this. The first reason is that the time it takes to warm up the HSM and reach the highest performance state. The second reason is the initialization

time for some processes, such as establishing a NTLS connection to the HSM, database connection, CA Key preparation, un-wrap key preparation. As a result the response times for first requests are significantly greater than for the following requests.

Table 2-8: Performance in Stockholm

Number of signing requests	Total time (ms)	Time per key pair (ms)
500	21,952	43.904
999	43,560	43.603
1,000	43,332	43.332

Table 2-9: Performance in Malmö

Number of signing requests	Total time (ms)	Time per key pair (ms)
9,998	206,455	20.649
9,999	203,902	20.392

The tests utilized both single threads and multithreads. The signing requests was processed in a total amount of time ranging from 43 ms to 6.6 ms per signing request in addition to the network delay. A major performance issue was expected to be the delay (or latency) when simultaneously processing a number of threads. Table 2-10 shows how the signing time varies with the number of threads, over the range from 1 to 50 threads. These results show that the batch processing of signing requests decreases from 43 ms to 6.6 ms as the number of threads increases from 1 to 50. However, at the same time, the processing time for a single signing requests is unstable (i.e., the variance in the time required per signing request increases). After conducting a number of different tests, we concluded that four threads were the optimal number of threads as this yielded a signing processing time of 6.9 ms *without* degrading the performance of individual signing request processing.

Table 2-10: Signing time per key pair with different numbers of simultaneous threads

Number of threads	Average time per signing request (ms)	Variance in signing times (ms)
1	43	0 ~ 20
2	15	0 ~ 30
4	6.9	0 ~ 30
10	6.8	0 ~ 100
30	6.7	0 ~ 500
50	6.6	0 ~ 700

We observed from the results shown in Table 2-10 that a stable signing processing time was not possible in the two devices tested when running with multithreads. However, as our objective was to

deploy our proposed solution to any HSM, even those that do not supporting multithreading, hence we performed the rest of our testing using only a single signing thread.

2.10.5 Signing process while performing unwrapping at the same time

Details of the implementation of proposed solution are given in Chapter 3. In this proposed solution, there is a requirement to load pre-generated keys from storage, un-wrap these keys, and initiate signing using the newly available keys. For this reason, the system's behavior needs to be analyzed when the signing process and the un-wrap key process are initiated at the same time. In the previous section we saw that the signing process had a stable execution time (taking a few milliseconds) when using less than four threads in the signing process. In this section, we evaluate parallel processing to un-wrap keys simultaneously with signing. Table 2-11 shows some of these test results. As can be seen from this table the impact of simultaneous signing and key unwrapping is minor, i.e. one or two millisecond when performing batches of key signing requests. This additional delay seems to be acceptable, especially when one considers that each unwrapping request makes available 2,000 new key pairs – in a time that is much shorter than the normal key generation time.

Table 2-11: Signing request time when simultaneously performing a key unwrapping process

Number of signing requests	Total time (ms)	Time per key pair (ms)
9,999	227,417	22.743
10,000	214,328	21.432

3 Implementation of the proposed solution

PKI solutions are great in demand for signing services (and many other applications). As we noted earlier, we expect that the demands on such services will increase, specifically that the peak arrival rate of signing requests could be up to 50 requests per second for a particular signing service. Conversely, the lowest arrival rate could be less than one, perhaps even zero for a period of a second. In this lowest rate of arrivals, we consider the system idle.

New services require a distinct RSA 2048-bit key pair for every signing request. According to the specifications of HSM used in our testing (see Table 2-3), it can create 1.7 such key pairs every second. As we can see this HSM could not meet the peak signing request of 50 signing requests per second. If we consider the peak hour on a given day of a year, then the number of key pairs required to meet a signing request rate of 50 requests / second would be:

60 minutes * 60 seconds * 50 signing request = 180,000 requests = = 180,000 key pairs

Unfortunately, in one hour the number of key pairs that can be generated by the HSM is only:

60 minutes * 60 seconds * 1.7 keys = 6,120

The above calculations illustrate the gap between the key generation rate and the expected peak signing request rate. In order to support this expected peak request rate we must either: (1) deploy a HSM that can generate key pairs at a much higher rate (leading to a much higher capital expenditure) or (2) we must generate key pairs in advance of the arrival of a signing request.

Fortunately, our HSM can generate RSA 2048-bit key pairs all day long at the rate of 1.7 such key pairs every second. In our proposed solution, we exploit the fact that the rate of arrival of signing requests varies over the months of the year, days of the month, and even hours within each day. During the time when the arrival rate of signing requests is low we accumulate generated but unused key pairs, we refer to this set of generated but unused keys as a key pool.

In this approach, we meet the peak demand by using keys from the key pool to perform the signing request. As long as we have a supply of key pairs from the key pool, we can sign at the maximum signing rate of the HSM (~1,200 signing operations per second) – rather than being limited to the key generation rate (~1.7 per second). A problem with this approach is that the HSM has only limited storage space, so we can only accumulate a limited number of key pairs. As can be seen from Table 2-2, even with a memory upgrade, the number of key pairs that can be stored ***within*** the HSM would be insufficient to meet the peak hour demands computed above (as 20,000 keys could be exhausted in 400 seconds if used at a rate of 50 per second).

To address the problem of limited memory capacity within the HSM, we needed to be able to store the pre-generated keys outside of the HSM, while still ensuring their security. In order to do this we wrap the key pair by encrypting the key pair and some metadata and then move this encrypted data out of the HSM to storage for future use. When the pool of available keys in the HSM falls below some threshold, we reload the key pairs into the HSM by retrieving wrapped keys from storage and unwrap them. Because the process of wrapping/unwrapping consumes time and we want to avoid limiting the performance of the HSM, we use symmetric encryption/decryption (specifically AES) using a key that is securely maintained inside the HSM. Since this decryption of a key pair takes less time than key generation we avoid the limitations of both the key generation rate and the limited memory capacity of the HSM.

From a key confidentiality point of view, we maintain the confidentiality of the key pool because a key pair is either stored securely inside the HSM or we maintain the confidentiality of externally stored key pairs by encrypting them using AES before exporting them outside of the HSM. Since the symmetric key(s) used for unwrap/wrap purposes are only stored inside the HSM it is infeasible to unwrap keys outside of the HSM. This means that all of the keys in the key pool are only accessible via the HSM.

When the HSM is idle or when the rate of signing requests is less than the key generation rate (~1.7 per second) the HSM will produce unused key pairs. These unused key pairs will be wrapped using a secret symmetric key, then transferred and stored outside the HSM.

Since we want to ensure that there are keys available from the key pool to perform signing operation, we use a thread to monitor the number of keys available in the pool and refill this key pool through loading wrapped keys from storage and unwrapping them using the secret symmetric key.

In order to monitor the performance of the HSM when signing we have added timestamps to the requests and responses. Each received signing request has a time stamp placed in the ''request Received Time Stamp" field when it is received. Each of these requests is placed in a FIFO queue. One or more signing threads service this queue. Every idle signing thread retrieves a request from the FIFO queue, takes a key pair from key pool, and uses this key pair to perform the signing process. When the signing thread starts to process a request it sets the "request Start To Process Time Stamp" field after retrieving the request from the queue request. When the signing thread finishes its processing it will set the ''request Finished Time Stamp" field. Using these timestamps, we can determine the queuing latency and processing times for every request.

3.1 Database

We have examined the time requirement for different storage media (as found in the section 2.9.7). The analyses of the advantages of the database over the flat file in the previous chapter lead to the choice of a database as the preferred storage media. This database offers higher transaction speeds as well as flexibility and increased information capacity for each of the objects. The database was used to store the exported keys and to store the time for generating the signature itself and the time to sign a certificate request. Using this database helped when extracting statistics and analyzing the system's performance.

Because the tests were done with different servers in the different locations and environments (Malmo and Stockholm) it was very useful to be able to test the solution with different databases. The UNIX based Ubuntu server in the Malmo center had a light-weight database called SQLite, while the Stockholm server runs on a Windows platform and SQL server 2008 was selected as the database for this site.

Figure 3-1 shows the table used for storing keys data into the database. The keys table contains five main columns: id ***prikey***, ***pubkey***, status and timestamp. These columns were used to support the key pool solution. For analysis purposes four additional columns were added to the keys table to store time stamps. A brief description of each of these columns is given in Table 3-1.

Column Name	Data Type
id	int
priKey	varbinary(MAX)
pubKey	varbinary(MAX)
keyGenerateStartTimeStamp	bigint
keyGenerateEndTimeStamp	bigint
keyGenerationDuration	bigint
keyUsedTimeStamp	bigint
status	int
timeStamp	timestamp

Figure 3-1: Database keys used for storage of data

Table 3-1: Description of the columns of the keys table

Id	a unique identifier
Prikey	binary value of the wrapped private key
pubkey	binary value of the public key
status	the status of the key pair (new, invalid, used)
timestamp	uniquely identifies the time stamp for each pair of the key expiration date.
keyGenerateStrtTimeStamp	when the key generation request was is sent to the HSM
keyGenerateEndTimeStamp	When the HSM sends the new generate key pair
keyGenerationDuration	time to generate the key pair.
keyUsedTimeStamp	when the key pair gets used

Figure 3-2 shows the table used for storing signature data into the database, while Table 3-2 briefly describes each of these columns. The signature table stores information about the signature requests and responses. The contents of the table that is crucial for the operation of the proposed solution include the *requestID*, *toBeSignedBytes*, *signatureBytes*, *certificateByte*, status, date, and timestamp. Additional columns were added to be consistent with key tables. For the individual single signature process, this information enables exact measurement of each step in the process. The timing information is used during the analysis phase. The tableis initially filled with hundreds of thousands of records in preparation for processing signing requests. The *requestID* is a unique identifier, generated by auto increment of a variable. The *toBeSignatureBytes* column filled with random byte arrays that represent the SAML signing request field *toBeSignedBytes*. The Status column is filled with a value of zero, which represents the state of processing having not (yet) started.

Column Name	Data Type
requestID	bigint
toBeSignedBytes	varbinary(MAX)
signatureBytes	varbinary(MAX)
requestReceivedTimeStamp	bigint
requestStartToProcessTimeStamp	bigint
requestFinishedTimeStamp	bigint
certificateBytes	varbinary(MAX)
status	int
date	datetime
timeStamp	timestamp
processTime	bigint
requestGetKeyPairTimeStamp	bigint
preGeneratedKeyPair	bit
TestID	varchar(MAX)
waitingTime	bigint

Figure 3-2: Columns of the signature table

Table 3-2: Description of the signature related columns

requestID	a unique identifier
toBeSignedBytes	the document digest
status	the current signature status in the records(represented as not used, received, signed, failed)
signatureBytes	the digital signatures
requestReceivedTimeStamp	When a request is received
requestStartToprocessTimeStamp	when the request starts to be processed
requestFinishedTimeStamp	when the request has been processed
certificateBytes	the certificate bytes
date	the request/processing date
timestamp	a unique time stamp for each record
processTime	the total time taken for the request to be signed
requestGetKeyPairTimeStamp	the time taken for a request to get a key pair
preGeneratedKeyPair	has the value “1” when the request used a pre-generated key pair
TestID	used to identifying the test being performed
waitingTime	the latency time (requestStartToprocessTimeStamp-requestReceivedTimeStamp)

3.2 Key Pool

The key pool is the main part of the implementation. It is responsible for two processes (shown in Figure 3-3). During the **idle time** of the HSM, the process begins by generating key pairs, then wraps the private keys by using symmetric AES keys stored inside the HSM, and saves the wrapped keys in a database. Later another process loads wrapped key pairs from the database, unwraps the private keys using the same symmetric AES keys stored inside the HSM, and the resulting key pair is made available in the HSM’s memory. The signing process can use the now available keys.

Each process has a single thread. The queue length is checked by the key generator and when there is no signing request in the queue, the process sends a key pair generation request. The process of generating the key will continue until either the defined maximum number of stored keys is reached or upon the receipt of new signing request.

The number of available keys in the pool is checked during key preparation. When the number of available keys falls below a defined minimum number of keys, then the thread starts to load a batch of keys from the storage media to be placed in the key pool after un-wrapping the private keys.

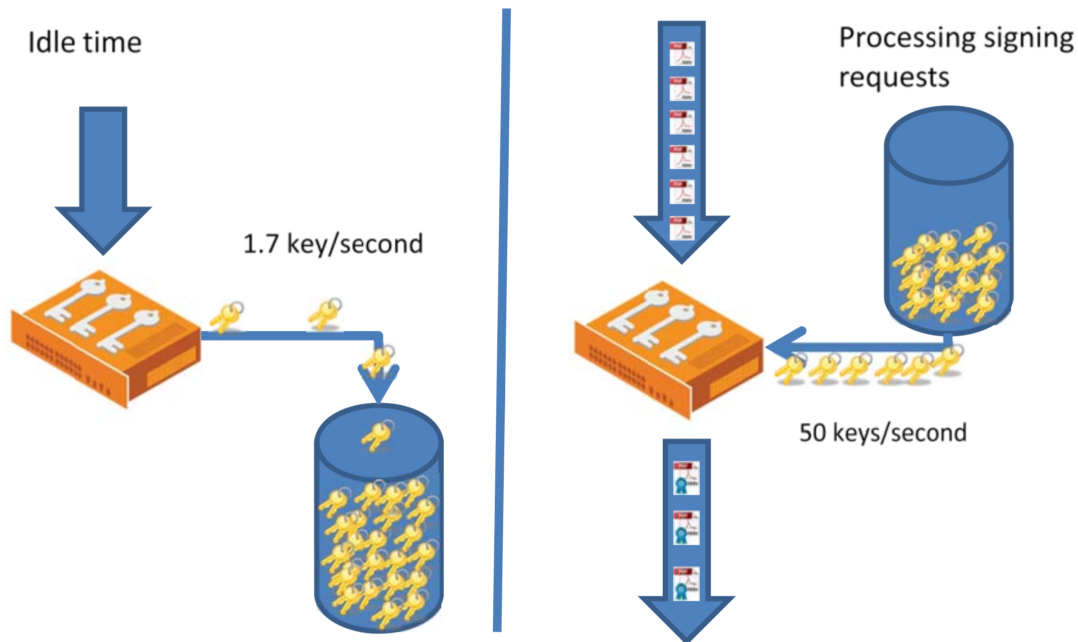


Figure 3-3: The idle time key generation process and the use of the key pool to process requests

3.3 Single HSM design

As it discussed in previous sections we have some main steps in key pool solution: generating the key pairs during the idle time or when the request rate is low, encrypt the key pairs with secret keys (symmetric keys), export and save the encrypted keys in the external storage. Then during the peak hours when the request rate is high encrypted key pairs are loaded from the storage media in to the HSM and the same secret keys will be used to decrypt the key pairs to be used in signing process.

Based on the system requirements and expected duration for peak hours we might need to generate and store millions of key pairs in advance. From a security point of view it is not a good idea to use one or few symmetric keys to encrypt all the key pairs. So it is good to generate and use new secret key for a bunch of key pairs. Since storage capacity of HSM's are limited so this cause another problem with storing and managing the secret keys. For solving this problem we can use well-known Secret Sharing mechanisms to export the secret keys and store them in the external storage as well. In this research we used a built in functionality of the Luna SA 1700 which is designed for similar purposes and called M of N PED* keys for "Cloning Domain" which is described in Section 3.5.

By using M of N keys for the "Cloning Domain" we can export the secret keys and store them in external storage while applying multi-person control over these keys. [29],[30]

* "*Luna PED is a PIN Entry Device, where PIN stands for Personal Identification Number. The PED works in conjunction with HSMs and backup tokens from SafeNet.*"[29] PED devices perform authentication processes by using PED keys. The M of N feature uses PED keys to split the secret over the N persons.

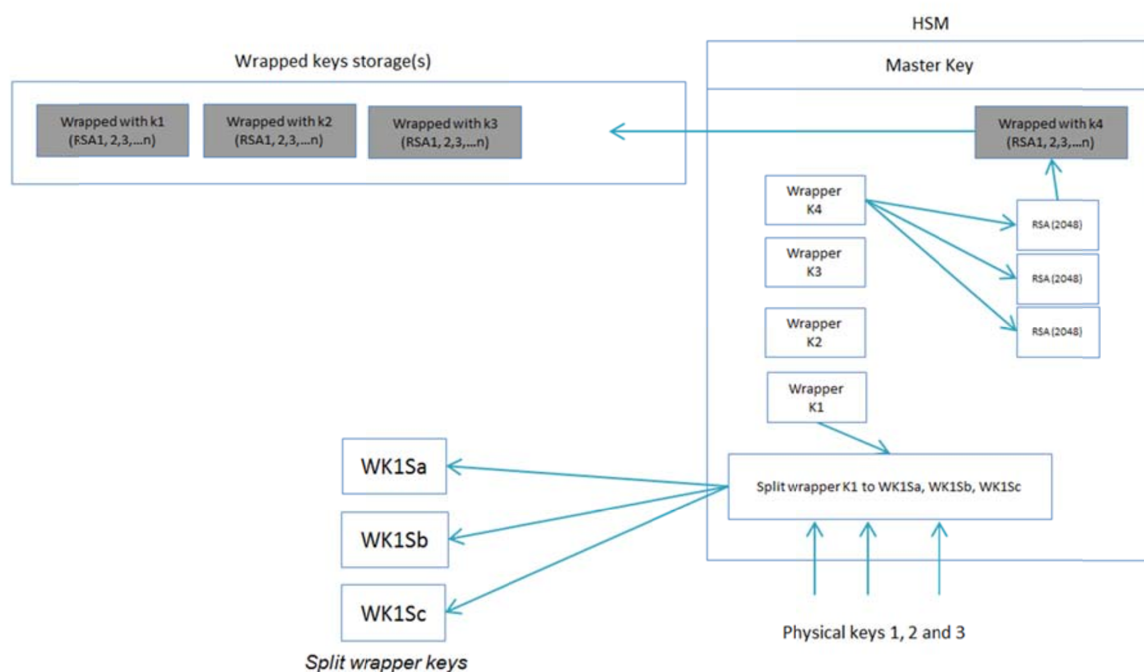


Figure 3-4: Single HSM design

3.4 Distributed design with more than one HSM

Scalability of the designed solution is very important because the demand for signing service is growing very fast and we must plan for the future when the system requirements will increase further. For example, if a new signature service customer asks for a peak rate of 100 signing requests per second, then we must be able to extend the solution by connecting more HSMs into the system. At the same time, from a security (availability) point of view we might need to distribute the system and install the HSM devices (main and backup) in different geographical locations. Therefore, we need a distributed design for our Key Pool solution.

As shown in Figure 3-5 we can extend or distribute the system by attaching new HSMs to the shared Key storage. First, we load secret keys into the new HSM by using the M of N functionality, then we load the encrypted keys from storage media in to the HSM. Then we can decrypt the wrapped key pairs by using the secret keys. Now we can immediately use these key pairs in signing processes. Depending upon the system design each HSM could play a specific role. For example, one HSM might only perform key generation thus filling the key pool to be used by other HSM(s) to perform the signing process.

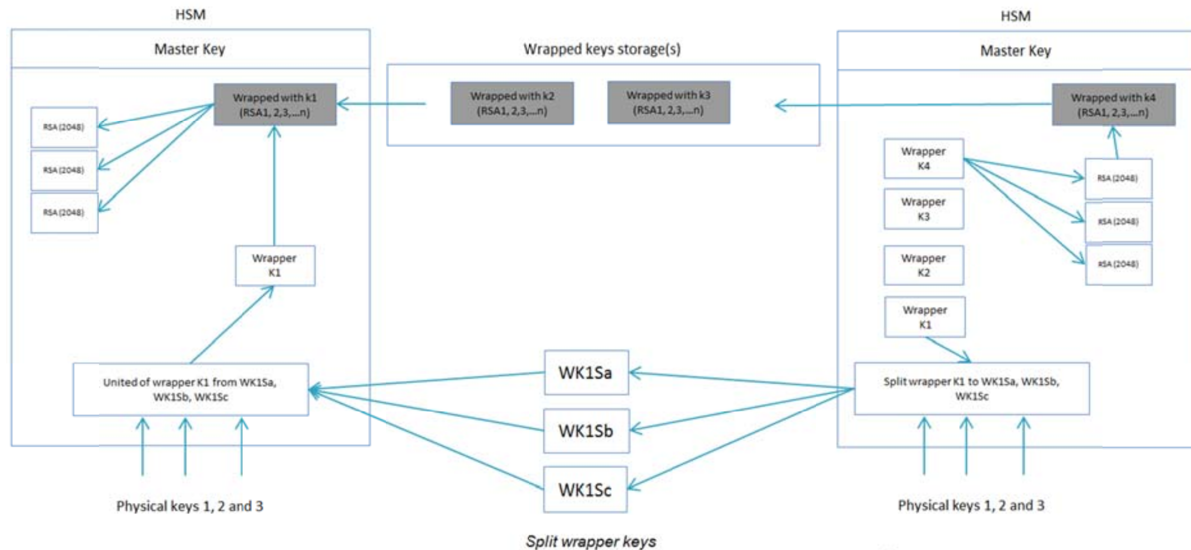


Figure 3-5: Design with multiple HSMs

3.5 Using Physical Keys in distributed systems (using multi HSM)

Secret sharing is a term which is used for splitting the secret object between N people in such a way that only a subset M of these N people need to put together their split parts in order to retrieve the unified secret object.[30] The aim is to enforce multi person control over the sensitive devices and data. This method is one of the built in functions of Luna SA 1700. This has been used for splitting the sensitive wrapper keys during the export and when moving the wrapper keys from a HSM outside or to another HSM. This functionality is very important from distributed system point of view. Secret Sharing can be used in different parts of the HSM configuration, but in this research we used the M of N functionality only in the “Cloning Domain”. The Cloning Domain is used for cloning or moving secret objects between HSMs and Backup devices.

The SafeNet Luna SA Manual describes this as:

“The M of N feature provides a means by which organizations employing cryptographic modules for sensitive operations can enforce multi-person control over access to the cryptographic module. The feature is available in all Luna SAs configured to use Trusted Path authentication – using the PIN Entry Device (PED) and PED Keys.

M of N involves a splitting of the authentication secret into multiple parts or splits. The shared secret is distributed (or “split”) among several PED Keys (“split-knowledge access control”). Every type of PED-administered HSM secret can be split when it is created: blue SO PED Key, black User/Partition Owner PED Key, ***red Cloning Domain PED Key****, orange Remote PED Vector key, purple Secure Recover Key.”[29]*

3.6 Sign request generator

A program called “*signRequestGenerator*” was written to perform the tests. The main responsibilities of the class is to generate a million random byte arrays with exactly the same length and form expected in the “*toBeSignedBytes”* field of a SAML signing request (see Appendix A) and to store those bytes within the signature table described in Section 3.1. The bytes represent the file digest that must be signed. In addition, the class is responsible for reading recordz from the signature table and creating a *signRequest* object and filling in parameters such as *TestID signRequestReceived Date* … . Moreover, this class places the object in the signRequestQueue, which is a FIFO list.

The need to perform the tests with distributions of requests makes use of a pre-defined number of signing requests and request generation rates. We have used 3000 requests starting at a rate of one signing request per second and increment the range up to 100 signing requests per second. The storage

of the time and information of each request enables the subsequent analysis of the system's operation characteristics. In addition, this testing process measures the latency difference over the distribution of requests. Table 4-1 in Chapter 4 describes the signing request distribution used for testing.

3.7 Sign Request handler

The *signRequestHandler* is a thread that checks in the *signRequestQueue* and keeps taking *signRequest* objects from the queue, fills in some parameters in the object, then signs the request by using a signing class (called Sign class). The java runnable handler class enables the use of multiple threading in the system. Tests were performed with from a one to 50 threads.

3.8 Signing

The Sign class retrieves the *signRequest* objects and prepares the key pair for the request in two ways after setting sometime parameters. When testing is performed with pre-generated keys the class gets a key pair from memory or from the key pool. In contrast, when testing without pre-generated keys the class simply sends a request to generate a new key pair to the HSM. The byte array is signed after getting the key pair in one of these ways, the signature bytes are added to the *signRequest* object, followed by destruction of the private key. The certification parameters and public key are delivered to the CA, after getting the certificate chains from the CA and storing the resulting information in the *signRequest* object (i.e., the new certificate and CA's certificate).

3.9 CA

The design made use of the BouncyCastle [24] code for handling CA certificate processes. The Sign class and CA deliver a CSR to the BouncyCastleCA. After setting the certificate parameters (such as Start Date, Expire Date, signature Algorithm, serial Number, …), then the CA creates a V3 certificate and signs the new public key.

4 Analysis

This chapter describes an evaluation of the proposed solution. It compares the proposed solution with the current performance of the specific HSM studied. The purpose of this comparison is to understand the benefits and limitations of the proposed solution as it might be applied to any HSM that is capable of exporting encrypted key pairs and importing encrypted key pairs.

Testing of the current HSM began by sending signing requests at the rate of one signing request per second. This load was increased every 100 requests by decreasing the time between requests by 100 ms, until reaching 100 requests per second, then decreasing the time between requests by 10 ms. Note that at 20 ms between requests the system reaches the desired maximum signing rate of 50 signing requests per second. This led us to a sequence of tests (shown in Table 4-1) that could be used for both the existing HSM and for the proposed solution.

Table 4-1: Sample of SigningRequest Generation following a specific distribution

From	To	Delay between requests (ms)	Requests per second
1	100	1000	1
101	200	800	1.25
201	300	700	1.43
301	400	600	1.66
401	500	500	2
501	600	400	2.5
601	700	300	3.33
701	800	200	5
801	900	100	10
901	1000	90	11.11
1001	1100	80	12.5
1101	1200	70	14.28
1201	1300	50	20
1301	1400	20	50
1401	**1500**	**10**	**100**
1501	1600	20	50
1601	1700	50	20
1701	1800	70	14.28
1801	1900	80	12.5
1901	2000	90	11.11
2001	2100	100	10
2101	2200	200	5

From	To	Delay between requests (ms)	Requests per second
2201	2300	300	3.33
2301	2400	400	2.5
2401	2500	500	2
2501	2600	600	1.66
2601	2700	700	1.43
2701	2800	800	1.25
2801	2900	900	1.11
2901	3000	1000	1

4.1 Key Generation on the fly

These initial experiments with the HSM showed that latency rapidly increased when the rate of signing requests exceed 1.7 signing requests per second, as this is the HSM's maximum key generation rate. Figure 4-1 portrays the signing delay experienced per request when testing with the arrival distribution shown in Table 4-1. It is very clear that the increasing latency in response time is very low (close to zero) when the request rate is less than the HSM's capability to generate key pairs on the fly and do the sign process. However, when increasing the request rates the latency goes up rapidly since HSM cannot generate key pairs at the same rate that requests arrives, thus a signing request must wait in the FIFO queue for a long time. For the last 400 requests, we can see a decrease in the latency because the request rate has once again less than the maximum rate at which the HSM can generate new key pairs. After this point number of the requests pending in the queue starts to decrease.

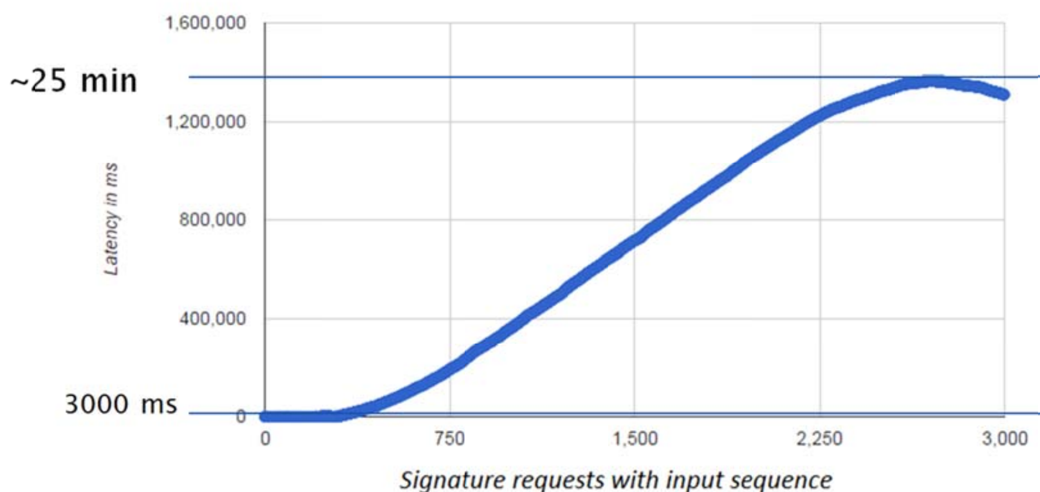


Figure 4-1: 3000 signing with key generation

Stability in the signing service is another important issue that needs to be addressed. The system should return to a normal state rapidly, even if it received a burst of requests at a higher request rate than the system can handle. Figure 4-2 shows the results of performed the previous test together with a extra 2000 signing requests sent at a low rate (1 per second). We can see in this figure that it takes a long time until the system returns to a normal state. As a result not only will requests sent during in the peak hours have a high latency but this earlier load will also affect requests waiting in the queue giving them a high latency as well.

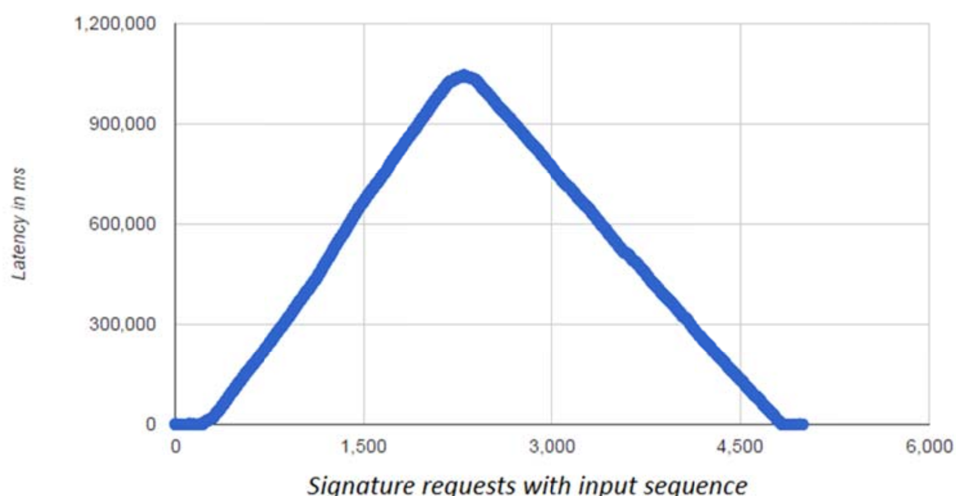


Figure 4-2: 3000 signing with key generation on the fly and 2000 additional signing with the one request per second

4.2 Pre-Generated keys

Figure 4-3 shows the latency distribution when using the same arrival signing request distribution and pre-generated keys. In this case, we see that as long as there are sufficient number of pre-generated keys in the HSM, there is no increase in the time required to process each signing request. The ability to continue signing at higher signing rates rapidly depletes the supply of available keys stored in the HSM. Once the set of keys in the HSM are exhausted, there is a momentary increase in the processing time. In short while the time to perform a single request increases rapidly as more and more requests are already in the queue ahead of each newly arriving request, this only lasts until the next set of pre-generated keys is retrieved from external storage. As soon as this new set of keys is available in the HSM the delay returns to a low value. Following this the delay continues to decrease with the decreasing request arrival rate of the test distribution, hence the time to process a signing request remains low.

Figure 4-3: 3000 sign with pre-generated keys

A comparison of the two tests can be summarized as:

- The key pool solution decreased the peak latency from 1,400,000 ms to 12,000 ms for the test arrival rate distribution; and
- The key pool solution enabled the system to return to lower delays much faster than the HSM could when not using the proposed solution.

4.3 Maximum size for the FIFO queue of requests

Figure 4-4 shows that when we set a maximum length for the FIFO to 50, that the queuing delay increases without bound – until the maximum FIFO queue capacity is reached at which point the system starts dropping requests. For these dropped requests, the response time is now infinite, but for the purpose of this graph these dropped requests are shown with a negative delay. Note that the system continues to process some requests, but most requests are being dropped. This test was performed both with and without using the key pool solution.

Figure 4-4 shows the result when the test was performed without using the key pool solution. It is readily apparent that the queuing delay increases rapidly with the increasing the request rate until reaching the arrival rate of 1.66 request per second. At this point the system has filled the FIFO queue and starting with request number 376 begins to drop requests. In this test 2073 out of 3000 requests were dropped.

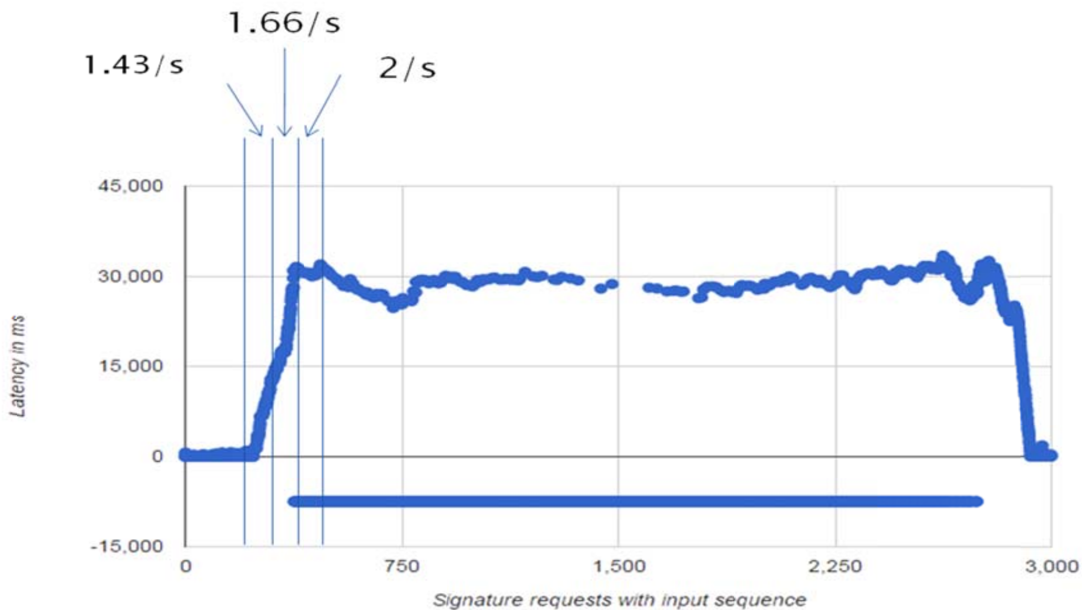


Figure 4-4: Latency when the maximum queue size is 50 *without* key pool solution (The vertical lines indicate the arrival rates of the test distribtuion.)

Figure 4-5 shows the results for the same test distribution performed when using the key pool solution. As it shown in this figure, requests are processes without a large delay when the arriving request rate is under or near to 50 requests per second. After exceeding this rate the queuing delay increases without bound until the FIFO queue is filled, then starting with request number 1537 the system starts to drop requests. In this test only 17 out of 3000 requests were dropped.

Comparing these two test result shows that key pool solution enables the system to process signing requests at higher rates, thus the system can operate for a longer time *without* filling the queue, hence avoiding dropping requests. Another important point is that the first dropped request number is number 1537, this occurred after the arrival rate reached 100 requests per second. Note that at this rate the system is no longer able to sign fast enough to keep up with the arriving requests – even though it has keys available. Note that in all but the 17 dropped requests that the delay was still within the bounds of 3000 ms, this shows that the system has greater tolerance to bursts than the system requirements.

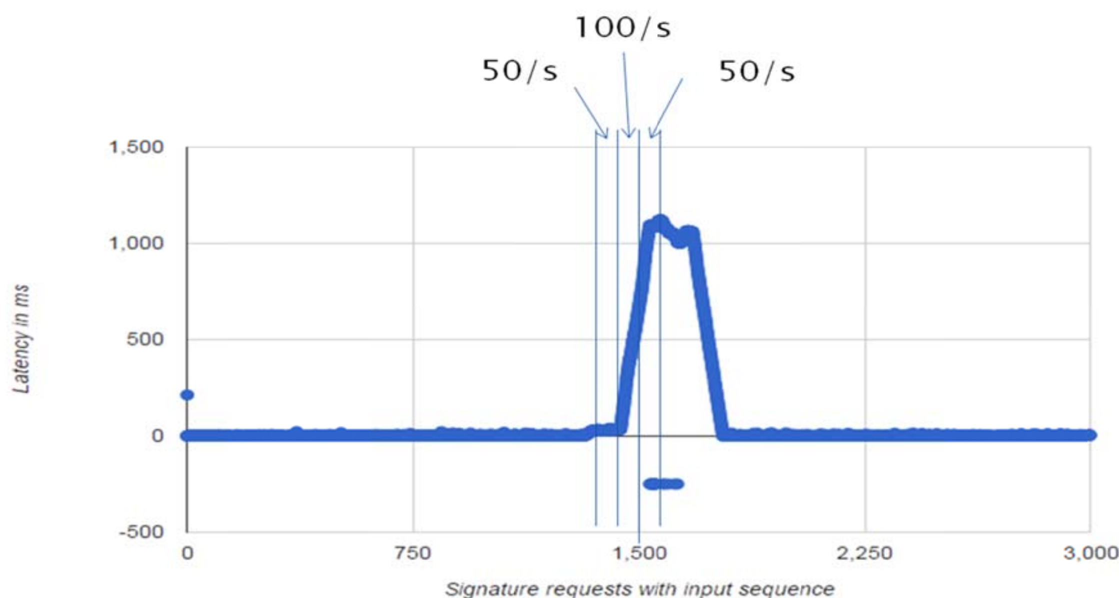


Figure 4-5: Latency when the maximum queue size is 50 with the key pool solution (The vertical lines indicate the arrival rates of the test distribtuion.)

4.4 Latency

The number of signing requests in the queue when a new request arrives and the signing processing time can be used to estimate the expected latency that will be experienced by a newly arriving signing request. By using the length of the signing request queue and the time to process each signing request, we can calculate the expected latency of this newly arriving request.

Moreover, we can use Little's Law (L = λW) to relate the number of signing requests waiting in the queue (L) to the average arrival rate (λ) and the average amount of time that requests have to wait to be serviced (W) – irrespective of the details of the arrival rate and the details of the service time.

In the Section 2.10.4, we determined the signing process time through measurements and labeled it 'Si'. This is the total response time for an individual signing request and includes two sign processes: signing the digest with private key of new key pair and signing the Certificate (new public key) by CA.

Here we introduce the term key preparation time (KP). This term represents the time needed to get a key to be used for signing. When a key is available from the HSM's internal RAM or storage, then the latency for a signing request is simply Si. When no keys are available inside the HSM, then if there are externally stored wrapped keys we have additional latency (the KP time) due to time needed to import and unwrap a key (see Section 2.10.5). If there is no externally stored wrapped key, then the latency for signing has to be increase by the key generation time (see Section 2.10.1). Given these terms, the signing latency can be described by the following equation:

$$Signing\ latency = \begin{cases} Si, \text{when a key is available} \\ Si + KP, \text{when a key is available externally} \\ Si + KGi, when\ a\ key\ is\ unavailable \end{cases}$$

As the maximum key generation rate is less than the signing rate there must be sufficient idle time when the rate of signing requests is less than the key generation rate in order to accumulate pre-generated keys. From the analysis above, we can see that during intervals when the average request arrival rate is less than or equal to the key generation rate (KG) that the size of the total key pool increases. When the arrival request rate is greater than the key generation rate (KG) the key pool decreases in size.

When the average arrival request rate is greater than the maximum signing rate, then the FIFO queue of pending requests will increase and hence the latency experienced by these requests will

increase – up to the maximum FIFO queue length – at which point requests will begin to be dropped. Given a maximum queue length (before requests are dropped), we can bound the maximum latency for successful requests to max_queue_length*Si = 3000 ms. Similarly we can bound the maximum arrival rate at which the system can stably operate based upon max_queue_length number of requests in 3000 ms (as after this point there is no ability to buffer additional requests). Thus it is possible for the system to operate for a bounded period of time even when requests arrive at a faster rate than the service rate for signing (as was shown in the previous section).

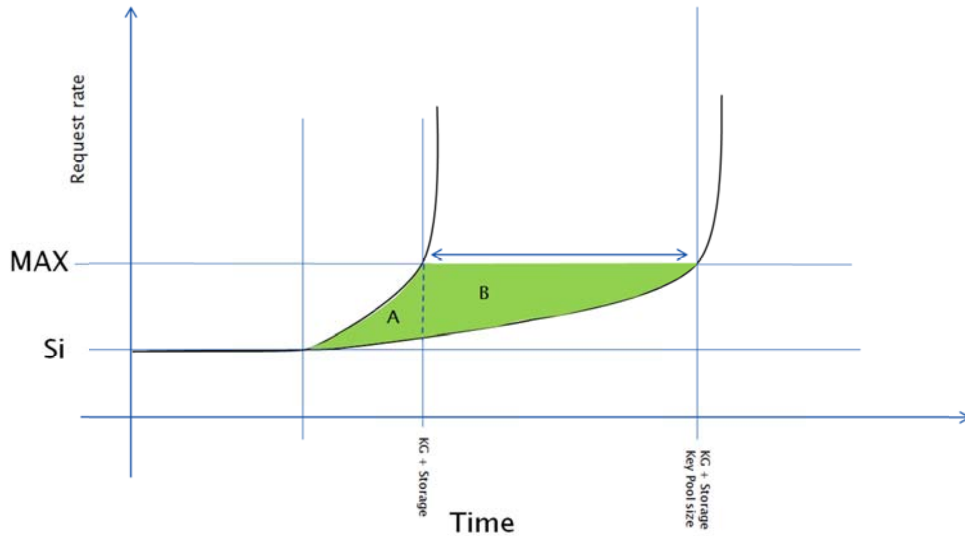


Figure 4-6: Time before first dropped request when using the Key Pool

Figure 4-7 illustrates that up to some arrival rate A, the signing latency is simply Si. After this, the latency increases up to the point at which the FIFO queue is filled, this gives a maximum latency (Max). After this, the latency is unbounded, as requests are dropped.

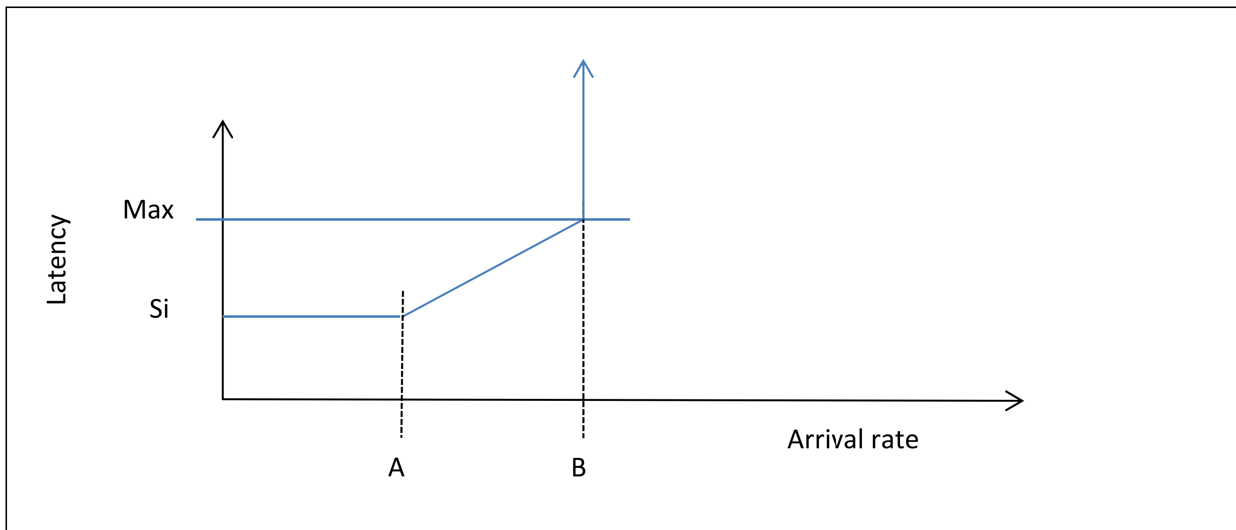


Figure 4-7: Latency versus arrival rate

Figure 4-8 shows how the FIFO queue space is a function of the arrival rate. When the average arrival rate is less than the signing service rate (C) the queue does not increase in length, hence the queue is on average empty. From this, we can see that the point A occurs then the arrival rate exceeds the signing rate.

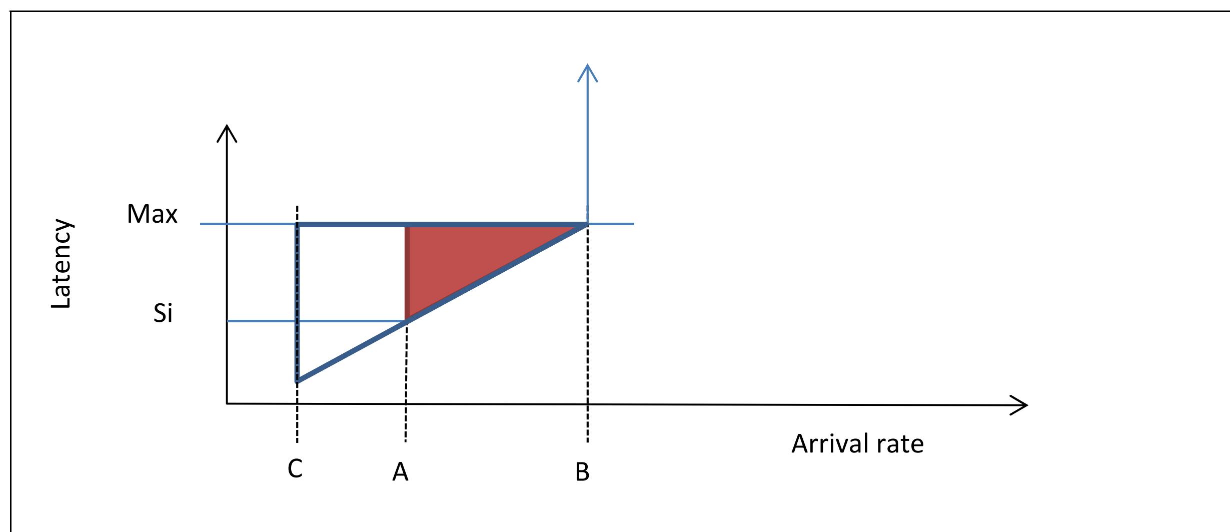


Figure 4-8: The triangle represents the available FIFO queue space and shows how it decreases with increasing arrival rates, at some point being exhausted.

Similarly, the integral of the difference between the request arrival rate and the key generation rate gives a bound on the total key pool size (i.e., the sum of the number of internal and external keys). The maximum size of the external storage can be used to bound the period of time that the average request rate can exceed the key generation rate (since in this bounded period of time the number of requests equals the maximum number of keys in the key pool – as after this there are no keys available, hence the service time will be once again limited by the KG time). Appendix B show test results demonstrating how the key pool can extend the stable response time of the system before reaching to the Max queue size and starting to drop requests.

4.5 Queue size and Latency Calculation in advance

Since the signing request rates changes over the different months, weeks, days, and hours, it would be good if we can estimate the expected latency in advance. This will enable us to understand the system's behavior in different situations *without* need to perform additional tests and timing measurements. A system designer can consider these results when dimensioning the system during the design phase.

4.5.1 Base Rate (BR)

Finding the Base Rate (BR) value is the initial step to calculate the expected latency. BR is the maximum request rate that the systems can respond to requests *without* the queue size increasing. Note that for a short period some request could be placed in the queue, but it should be removed from the queue very quickly in order that the long term behavior is that of a stable system with low latency and a queue size close to zero.

To examining BR we performed some tests and check the results. Since this test was performed in the Stockholm office the base signing time is close to 40ms (as discussed in Section 2.10.4). We started our testing at 25 requests per second (1000 ms /40 ms = 25). Figure 4-9 shows that the result for this test are quite stable. The next test increased the rate to 26.3 requests per second. Figure 4-10 show that the system is still quite stable. The final test was performed at 27 requests per second. The result of

this final test (shown in Figure 4-11) shows that queue size and latency are both starting to increase. Based upon these results we take the rate of 26.3 request per second as our BR.

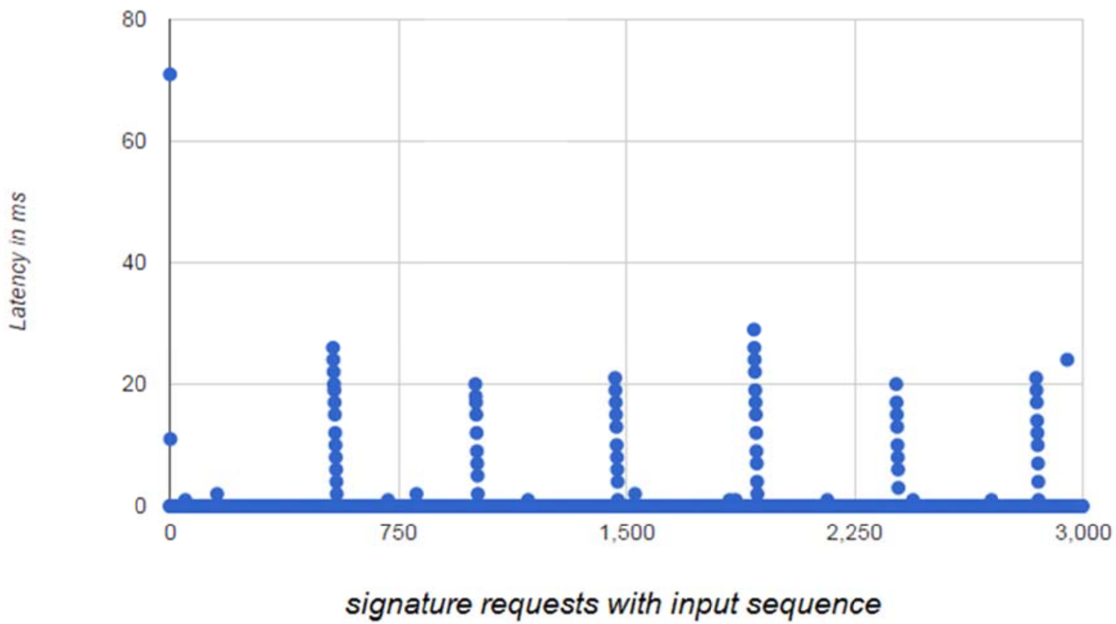


Figure 4-9: Processing of 3000 signing requests at 25 requests per second

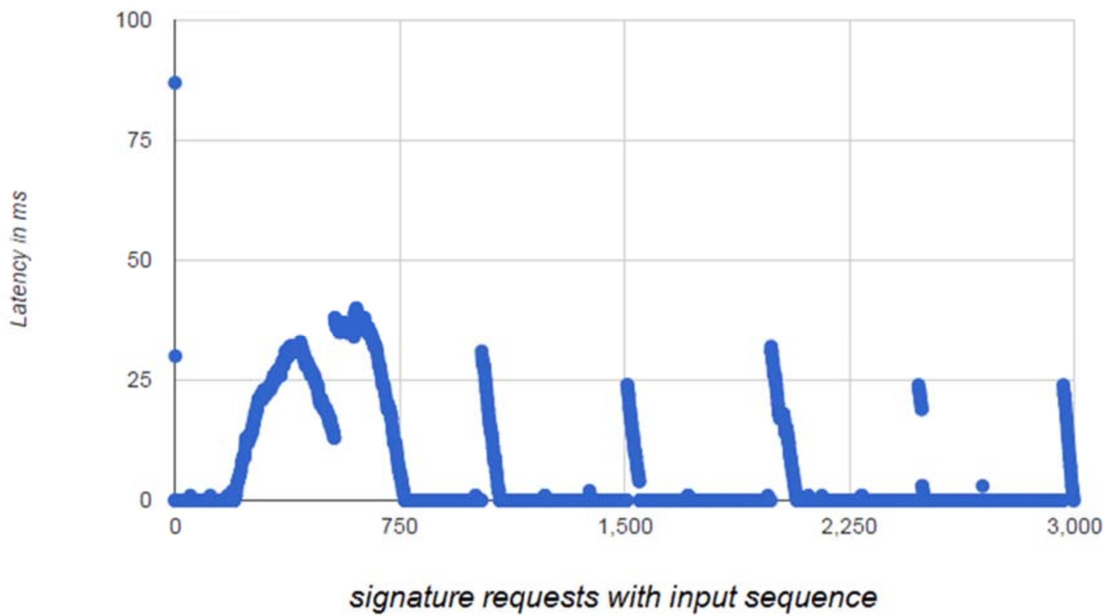


Figure 4-10: Processing of 3000 signing requests with at 26.3 requests per second

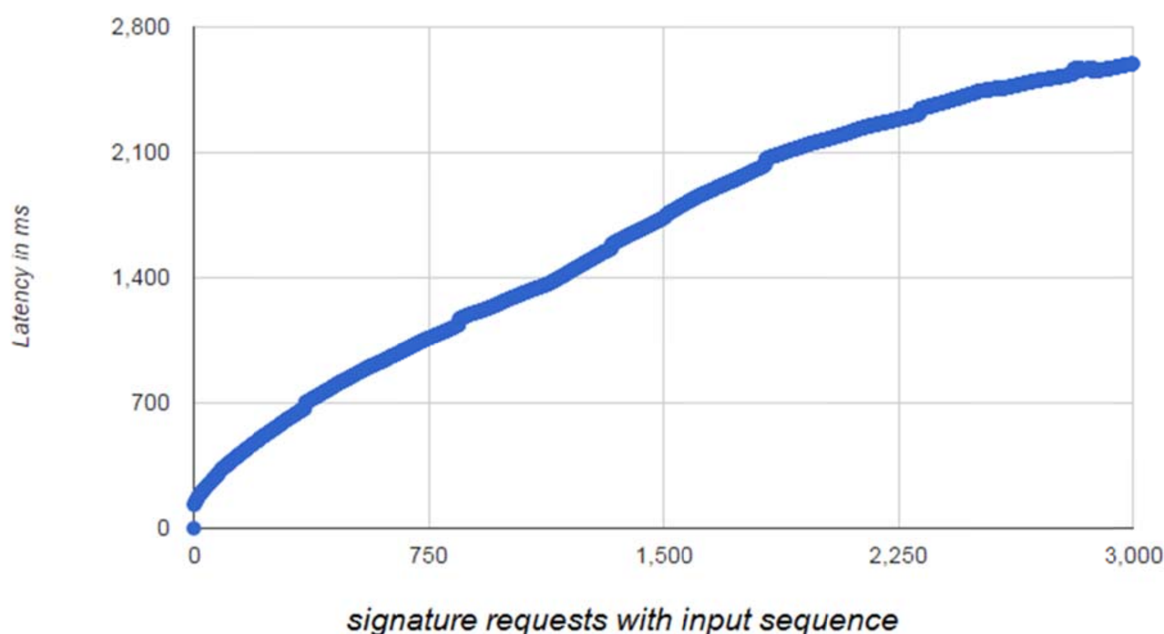


Figure 4-11: Processing of 3000 signing requests at 27 requests per second

4.5.2 Queue Size (Q)

Knowing BR we can calculate the queue size after a specific time (T) if we receive request at a higher rate (HR).

Q= (HR - BR) * T

For instance if our system receives requests at a rate or 40 requests per second for 100 seconds, then at the end of this 100 seconds we expect to have following number of requests in the queue:

Q = (40-26.3) * 100 = 1370

If this value is greater than the actual maximum queue size, then we can compute how many of these requests would be dropped.

4.5.3 Key Preparation (KP) time

In this context, Key Preparation time is the time that takes to have a key pair ready in the HSM to use in the signing process. KP time changes in different situations. If we have key pairs available inside the HSM (in storage or RAM) then KP is equal to zero. In the key pool solution KP represents the time that it takes the system to load and decrypt a key pair. Note that in the key pool solution if we perform loading and decryption in parallel with the signing process then KP is also zero. Only if we need to generate a key pair is the KP time non-zero, in this case it will be equal to the key generation time of the HSM, for example for the Luna SA 1700 this will be 588ms.

4.5.4 Latency calculation (L)

Knowing the BR, Q, KP, and Si (see Section 2.10.4) now we can calculate the latency for a request that we expect to receive at the time T by using following formula: L = (KP + Si) * Q.

4.6 Reliability / validity Analysis

We performed serveral tests in the Stockholm office to examine the values of BR and Si. Based upon a number of tests we determined that BR = 26.3 requests per second and Si = 38 ms. Next three signing tests were performed and the results compared with the results of using the queue and latency formulas. By comparing these results, we can assess the accuracy of our formulas.

Table 4-2 shows the result for the three tests with different request rates and timing measurements, the actual queue size and latency, and the calculated values for both latency and queue length. Based upon these results we see that the calculation of the queue length is very close to the actual queue length (Q), while the actual latency is within 2% of the calculated latency. Interestingly the actual latency is all cases is slightly less than the calculated latency.

Table 4-2: Latency calculation vs actual test results

Test ID	HR (sign/ s)	Process duration (s)	Calculated Q	Actual Q	Latency Calculation (s)	Actual Latency (s)	Accuracy
1	40	75.001	1027.5	1028	39.045	38.565	98.76 %
2	50	59.990	1422	1422	54.036	53.266	98.66 %
3	100	29.995	2211	2217	84.018	83.314	99.16 %

Calculations:

Q = (CR-BR) * T

Q = (40 – 26.3) * 75.001 = 1027.5

Q = (50 – 26.3) * 59.990 = 1422

Q = (100 – 26.3) * 29.995 = 2211

Latency = (KP + Si) * Q

Latency = (0+38 ms) * 1027.5 = 39045 ms = 39.045 s

Latency = (0+38 ms) * 1422 = 54036 ms = 54.036 s

Latency = (0+38 ms) * 2217 = 84018 ms = 84.018 s

5 Conclusions and Future work

This chapter presents some conclusions and a summary of the thesis project. It reviews the outcome of the tests. Section 5.2 identifies some of the limitations inherent in the design as realized via the test sessions. The chapter concludes with some of the insights gained through personal reflections.

5.1 Conclusions

The research goals were met because the response time for the HSM could be improve using pre-generated keys and using those keys via a "Key Pool". The design could support up to 50 signing requests per second without adverse latency effects even with a single thread. As discussed in Section 2.10.4 a minimum of 20 ms is required to perform the full signing process, hence 50 signatures per second is the maximum capacity of the system when excluding the key preparation process. The proposed solution optimizes the HSM performance and enables it to operate within the bounded response time at the desired peak signing rate. Enabling multi-threading reduces the response time from 742 ms to 20 ms which translates to a 37 times faster operation. Performance can be further improved by applying multi-threading, but the implementation of this is left for future work.

The response time for the sign process was reduced within the HSM by multi-threading, but there was an increased latency for each individual signing process. To sign each of 3,000 requests with a single thread takes 20 ms without additional latency or queuing, hence a total of 60,000 ms. When multi-threading, it took a total of 6,000 ms to complete all 3,000 requests. Hence, there is a large reduction it the total completion time when a multi-threaded approach is used.

When performing the digital signature process using a key pool the individual steps are key generation, wrapping, insertion into the database, reading from the database, unwrapping, and signing a document with the new private key. We isolated each of these steps and performed tests that enabled us to measure the performance of each step in the complete process. Analysis of this data enable us to identify the critical step(s), for which we needed improved performance.

Noting that the processing could be implemented as parallel processes, further improvements could be made after isolated the key generating and signing processes. The tests parallelized networking, logging in, and database transaction in order to find out how these affected the overall processing time. It was crucial to isolate the timing of each process in order to understand where the bottlenecks were.

If I was to do this thesis project again, I would focus on minimize the communication delay and variations in order to get better results, while making more measurements. Additionally, using two or more different brands of HSMs (in the same class) would be a good way to test and compare these different brands of HSMs.

The analyses of the test results lead to the conclusion that the proposed solution increases the performance for the HSM while improving security. There are inherent disadvantages of the key pool with respect to a single HSM. These disadvantages include increased communication delay, due to the communication that occurs between the HSM, the application server, and the database. It is worth noting that this delay was very small, but not negligible. Moreover, the lack a standard method for implementing the proposed key pool solution could cause reduced availability – as the system's availability now depends upon the availability of the database (as this stores the key pool). This suggests that for future work there is a need for a high availability database for use with the key pool. A poor implementation of the key pool could reduce the confidentiality of the keys. To achieve better performance and increase reliability it is strongly suggested that HSM vendors introduce key pool functionality into their HSMs in an interoperable (standardized) manner. This would enable users to benefit from this built-in functionality, while also enabling them to mix and match combinations of HSMs from different vendors.

As governments have changed the requirements for digital signature services - forcing the service providers to use a unique key pair for each signing process, hence sooner or later these changes will

force service providers to use a key pool or similar solution in order to be able to handle the expected peaks in signing request rates. In such a competitive market HSM devices will gain market share if offer a built-in key pooling solution to their customers. This is expected to lead to the situation where if a single vendor introduces a key pool solution, all other vendors will be forced to introduce a similar solution - otherwise they will no longer be competitive.

5.2 Limitations

This proposed solution only works with export enabled HSMs. In addition, these HSMs do not allow the users of the platform to export objects that are sensitive, including private keys regardless of the fact that the key is wrapped using a secure method. The internal memory capacity of the HSM can be used to store sensitive private key objects.

5.3 Future work

A considerable amount of work remains, including:

- Further research study on Private Keys is required. A Private Key in RSA format has some additional data which is very sensitive. A good solution would be to remove the private exponent and sensitive parts of the private key from the file and wrap and export only those parts after wrapping. Another solution would be keep these sensitive parts of the private key inside the HSM, while wrapping and exporting the rest. Keeping all the sensitive parts inside the HSM will increase the level of security, but at the cost of limiting the size of the key pool. Performance analysis of these ideas will be a challenging part of such a solution.
- Using a semi-automatic machine learning mechanism could be used to analyze request rates over the time in order to better anticipating the future request rate distribution.
- Another method to improve the integrity and confidentiality of the solution would be to apply a linking schema to the time-stamps (attached to the exported keys)[31].
- Using a Programmable HSMs such as Safenet's Luna SP to implement the key pool solution *inside* the HSM could minimize communication delays.

5.4 Reflections

The cost to the customers of a signing service provider can be reduced as the number of expensive devices required to support a given peak load is decreased with the proposed solution. This reduction in cost could potentially extend the use of such a service to many more applications and people. The reduction in the number of HSMs require not only affects the capital expenditure and operating costs of the signing service provider, but also leads to environmental benefits – as described further below.

A fast digital signature service combined with a reduction of service price enables the system to serve many people on a daily basis. This usage can be for personal use or business communication, transactions, or trades; while providing users with high reliability with confidentiality.

The environmental benefit that accompanies the system is large. The proposed solution is “green” in that reduces the number of devices used (in this case) by an order of magnitude. As a result, the material used is minimized, resources conserved, and time and energy in the production of each device is reduced. Fewer devices results in a less electricity being used to operate them, hence the result is a cumulative reduction in the amount of energy needed. The numbers of devices that will need to be recycled are also reduced, thus avoiding further degradation of the environment.

This project has shown the feasibility of the proposed solution, but future users will have to wait for the key pool concept to be introduced into commercial products – before most of the advantages can be exploited.

References

[1] "Ramavtal eID2008." [Online]. Available: http://www.elegnamnden.se/fragorsvar/faq/vadskakommunernagoranarderaselegitimationsavtalgarut2012behovsennyupphandling/ramavtaleid2008.4.71004e4c133e23bf6db800051958.html. [Accessed: 09-Nov-2014].

[2] Utredningen om bildande av en e-legitimationsnämnd., *E-legitimationsnämnden och svensk e-legitimation: betänkande - ISBN 9789138235072 9138235072*. Stockholm: Fritze, 2010.

[3] Roberth Lundin, "EFST Underskriftstjänst och hantering av nycklar i tjänsten v2." Cybercom Group AB, 26-May-2014.

[4] "Internet Explorer on Windows 8.1: One browser, two experiences" [Online]. Available: http://msdn.microsoft.com/en-us/library/ie/hh771832(v=vs.85).aspx#plugins. [Accessed: 22-Dec-2014].

[5] "Get ready for plug-in free browsing (Internet Explorer)." [Online]. Available: http://msdn.microsoft.com/en-us/library/ie/hh968248(v=vs.85).aspx. [Accessed: 22-Dec-2014].

[6] "NPAPI Plugins - Google Chrome." [Online]. Available: https://developer.chrome.com/extensions/npapi. [Accessed: 20-Dec-2014].

[7] "Saying Goodbye to Our Old Friend NPAPI," *Chromium Blog*. .

[8] "Welcome to ebIX." [Online]. Available: http://www.ebix.org/content.aspx?ContentId=1009&SelectedMenu=62. [Accessed: 17-Dec-2014].

[9] "XML Advanced Electronic Signatures (XAdES)." [Online]. Available: http://www.w3.org/TR/XAdES/. [Accessed: 17-Dec-2014].

[10] "Om Sambi | Sambi," 19-Nov-2014. [Online]. Available: https://www.sambi.se/om/. [Accessed: 19-Nov-2014].

[11] M. Bishop, *Introduction to computer security - ISBN 0321247442*. Boston: Addison-Wesley, 2005.

[12] Johan Ivarsson and Andreas Nilsson, "A Review of Hardware Security Modules Fall 2010," Certezza.

[13] Sokratis K.Katsikas, Stefanos Gritzalis, and Javier Lopez, *Public Key Infrastructure (Paper Collection) - ISBN 3540222162*. Springer, 2004.

[14] Ravneet Kaur and Amandeep Kaur, "Digital Signature," presented at the IEEE - International Conference on Computing Sciences, 2012.

[15] A. Müller, H. Schröder, and L. von. Thienen, *Lean IT-Management was die IT aus Produktionssystemen lernen kann - ISBN 978-3-8349-2910-5*. Wiesbaden: Gabler, 2011.

[16] K. H. Brown, "Security requirements for cryptographic modules," *Fed. Inf. Process. Stand. Publ.*, pp. 1–53, 1994.

[17] Lynn Margaret Batten, *Public key cryptography applications and attacks*. Hoboken, N.J.: John Wiley & Sons, 2013.

[18] M. Y. Rhee, *Wireless Mobile Internet Security (2nd Edition)*. Somerset, NJ, USA: John Wiley & Sons, 2013.

[19] "X.509 certificates." [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/security/cert3.html. [Accessed: 24-May-2014].

[20] "A Layman's Guide to a Subset of ASN.1, BER, and DER." [Online]. Available: http://luca.ntop.org/Teaching/Appunti/asn1.html. [Accessed: 04-Jun-2014].

[21] C. Steel, *Core security patterns: best practices and strategies for J2EE, Web services, and identity management - ISBN 0131463071*. Upper Saddle River, NJ: Prentice Hall PTR, 2006.

[22] K. Roebuck, *Security assertion markup language (SAML): high-impact strategies - what you need to know: definitions, adoptions, impact, benefits, maturity, vendors - ISBN 9781743046258 - 1743046251*. Milton Keynes: Lightning Source, 2011.

[23] "Java SE Security." [Online]. Available: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html. [Accessed: 24-May-2014].

[24] "bouncycastle.org." [Online]. Available: https://www.bouncycastle.org/documentation.html. [Accessed: 25-May-2014].

[25] "Hardware Security Modules (HSMs) | SafeNet Encryption & Key Security," *safenet-inc.com*. [Online]. Available: http://www.safenet-inc.com/Products_V2/tier2.aspx?id=8589945123. [Accessed: 27-May-2014].

[26] Jim Attridge, "SANS Institute InfoSec Reading Room." 14-Jan-2002.

[27] "SafeNet Luna SA 5 Price list."

[28] "Luna SA Network-Attached HSM | Hardware Security Module | SafeNet," *safenet-inc.com*. [Online]. Available: http://www.safenet-inc.com/Products_V2/tier4.aspx?id=2147483853. [Accessed: 27-May-2014].

[29] "SafeNet Luna SA Manual." SafeNet, 12-Mar-2014.

[30] C. Blundo, A. De Santis, R. De Simone, and U. Vaccaro, "Tight bounds on the information rate of secret sharing schemes," *Des. Codes Cryptogr.*, vol. 11, no. 2, pp. 107–110, 1997.

[31] S. Haber and W. S. Stornetta, *How to time-stamp a digital document*, vol. 3. Jornal of Cryptology, 1991.

Appendix A: SAML Signing Request / Response

Signing Request:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- [Mandatory] Sign request root element (namespace OASIS DSS). The Profile
attribute will always have the value of the EID 2.0 DSS implementation profile. -->
<dss:SignRequest xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema"
                 RequestID="da3fb4d0268a41f9b02c5feda32826de"
                 Profile="urn:comfact:cgi:dss:sscd:1.0:profile">
  <dss:OptionalInputs>
    <!-- [Mandatory] The SignRequester element (namespace EID 2.0 DSS EXTENSIONS)
describes the customer (requesting service provider) making the sign request. -->
    <eid2:SignRequester Format="urn:oasis:names:tc:SAML:2.0:nameid-format:entity"
xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-
ext/ns">https://testorg.se/sign</eid2:SignRequester>
    <!-- [Mandatory] The RequestedSignatureAlgorithm element (namespace EID 2.0 DSS
EXTENSIONS) specifies which signature algorithm to use. -->
    <eid2:RequestedSignatureAlgorithm
xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-
ext/ns">http://www.w3.org/2001/04/xmldsig-more#ecdsa-
sha256</eid2:RequestedSignatureAlgorithm>
    <!-- [Mandatory] The CertRequestProperties element (namespace EID 2.0 DSS
EXTENSIONS) CertType attributes describes what type of certificate to generate. -->
    <eid2:CertRequestProperties xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-
ext/ns" CertType="QC/SSCD" />
    <!-- [Mandatory] The SAMLAuthContext element (namespace saci, SAMLAuthContext )
describes all certificate fields attributes to be used in the certificate -->
    <saci:SAMLAuthContext xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:saci="http://id.elegnamnden.se/auth-cont/1.0/saci"
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">
      <saci:AuthContextInfo ServiceID="ca4104354a972bce"
AssertionRef="_0ec54156665f47e7ba4c961b296f94ae" IdentityProvider="https://m00-mg-
local.testidp.funktionstjanster.se/samlv2/idp/metadata/0/0"
AuthenticationInstant="2013-06-24T13:30:57.521+02:00"
AuthnContextClassRef="http://id.elegnamnden.se/loa/1.0/loa3"/>
      <saci:IdAttributes>
        <saci:AttributeMapping Type="rdn" Ref="2.5.4.5">
          <saml:Attribute FriendlyName="Personnummer"
Name="urn:oid:1.2.752.29.4.13">
            <saml:AttributeValue
xsi:type="xsd:string">195006262546</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
        <saci:AttributeMapping Type="rdn" Ref="2.5.4.6">
          <saml:Attribute FriendlyName="Land" Name="urn:oid:2.5.4.6">
            <saml:AttributeValue xsi:type="xsd:string">SE</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
        <saci:AttributeMapping Type="rdn" Ref="2.5.4.42">
          <saml:Attribute FriendlyName="Förnamn" Name="urn:oid:2.5.4.42" >
            <saml:AttributeValue
xsi:type="xsd:string">Valfrid</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
        <saci:AttributeMapping Type="rdn" Ref="2.5.4.3">
```

```
          <saml:Attribute FriendlyName="Användarnamn"
Name="urn:oid:2.16.840.1.113730.3.1.241">
            <saml:AttributeValue xsi:type="xsd:string">Valfrid
Lindeman</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
        <saci:AttributeMapping Type="rdn" Ref="2.5.4.4">
          <saml:Attribute FriendlyName="Efternamn" Name="urn:oid:2.5.4.4">
            <saml:AttributeValue
xsi:type="xsd:string">Lindeman</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
        <saci:AttributeMapping Type="san" Ref="1">
          <saml:Attribute FriendlyName="E-post"
Name="urn:oid:0.9.2342.19200300.100.1.3">
            <saml:AttributeValue
xsi:type="xsd:string">vli@example.com</saml:AttributeValue>
          </saml:Attribute>
        </saci:AttributeMapping>
      </saci:IdAttributes>
    </saci:SAMLAuthContext>
  </dss:OptionalInputs>
  <dss:InputDocuments>
    <dss:Other>
      <!-- [Mandatory] The SignTasks element (namespace EID 2.0 DSS EXTENSIONS)
provides input data to the signature generation process. -->
      <eid2:SignTasks xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-ext/ns">
        <eid2:SignTaskData SignTaskId="SignTask1" SigType="XML" AdESType="None">
<eid2:ToBeSignedBytes>PD94bWwgdmVyc2lvbj0iMS4wIj8+DQo8U2lnbmVkSW5mbyB4bWxuczp4c2k9I
mh0dHA6Ly93d3cudzMub3JnLzIwMDEvWE1MU2NoZW1hLWluc3RhbmNlIiB4bWxuczp4c2Q9Imh0dHA6Ly93
d3cudzMub3JnLzIwMDEvWE1MU2NoZW1hIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC8wOS94bWx
kc2lnIyI+DQogIDxDYW5vbmljYWxpemF0aW9uTWV0aG9kIEFsZ29yaXRobT0iaHR0cDovL3d3dy53My5vcm
cvVFIvMjAwMS9SRUMteG1sLWMxNG4tMjAwMTAzMTUiIC8+DQogIDxTaWduYXR1cmVNZXRob2QgQWxnb3Jpd
GhtPSJodHRwOi8vd3d3LnczLm9yZy8yMDAxLzA0L3htbGRzaWctbW9yZSNyc2Etc2hhMjU2IiAvPg0KICA8
UmVmZXJlbmNlPg0KICAgIDxUcmFuc2Zvcm1zPg0KICAgICAgPFRyYW5zZm9ybSBBbGdvcml0aG09Imh0dHA
6Ly93d3cudzMub3JnLzIwMDAvMDkveG1sZHNpZyNlbnZlbG9wZWQtc2lnbmF0dXJlIiAvPg0KICAgIDwvVH
JhbnNmb3Jtcz4NCiAgICA8RGlnZXN0TWV0aG9kIEFsZ29yaXRobT0iaHR0cDovL3d3dy53My5vcmcvMjAwM
S8wNC94bWxlbmMjc2hhMjU2IiAvPg0KICAgIDxEaWdlc3RWYWx1ZT5VYVFVNkw4T3BFQTNJdmU2c2ViOWR0
V0lZNnZwd0VKU3JUVFlUZWRBVEU0PTwvRGlnZXN0VmFsdWU+DQogIDwvUmVmZXJlbmNlPg0KPC9TaWduZWR
JbmZvPg==</eid2:ToBeSignedBytes>
        </eid2:SignTaskData>
        <eid2:SignTaskData SignTaskId="SignTask2" SigType="PDF" AdESType="None">
<eid2:ToBeSignedBytes>MWkwGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEHATAcBgkqhkiG9w0BCQUxDxcNM
TMwNzEwMTQxNjA0WjAvBgkqhkiG9w0BCQQxIgQgVntedHsL4eHGMfvybmqhalkprh3iJEzQGMer1/nCIKo=
</eid2:ToBeSignedBytes>
        </eid2:SignTaskData>
      </eid2:SignTasks>
    </dss:Other>
  </dss:InputDocuments>
</dss:SignRequest>
```

Signing Response:

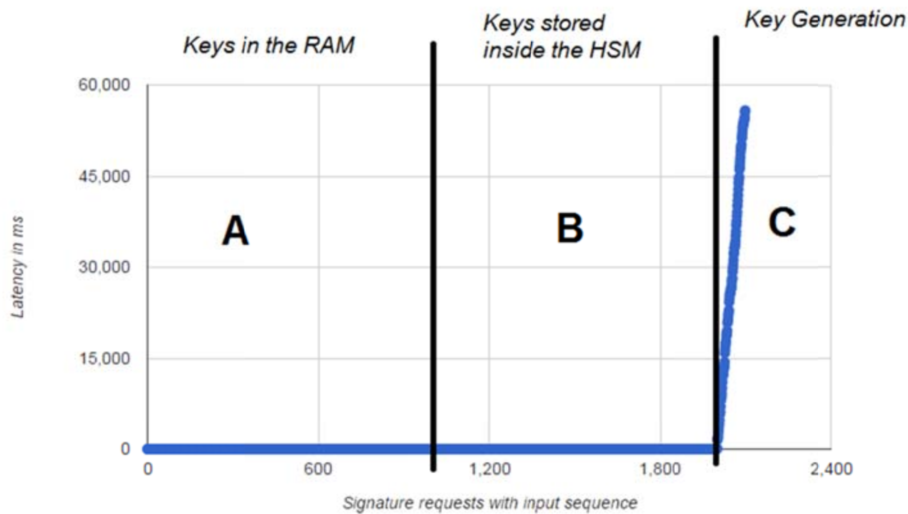
```
<?xml version="1.0" encoding="utf-8"?>
<!-- [Mandatory] Sign response root element (namespace OASIS DSS). The values of
the RequestID and Profile attributes must match the ones in the request. -->
<dss:SignResponse RequestID="da3fb4d0268a41f9b02c5feda32826de"
Profile="urn:comfact:cgi:dss:sscd:1.0:profile"
xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema">
  <!-- [Mandatory] The Result element (namespace OASIS DSS) signals whether the
request was successfully completed. -->
  <dss:Result>

<dss:ResultMajor>urn:oasis:names:tc:dss:1.0:resultmajor:Success</dss:ResultMajor>
  </dss:Result>
  <dss:OptionalOutputs>
    <!-- [Mandatory] The SignatureCertificateChain element (namespace EID 2.0 DSS
EXTENSIONS) holds the full certificate chain. -->
    <!-- Note that this element is marked as mandatory, but that of course only
holds if the service operation succeeded. -->
    <eid2:SignatureCertificateChain
xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-ext/ns">
      <eid2:X509Certificate>MIIPkjCCDnqgAwIBAgIQGGe…F63kjsYW1Ex8V2/+
</eid2:X509Certificate>
      <eid2:X509Certificate>MIIFtDCCA5ygAwIBAgIJAIGkz…lmj+ </eid2:X509Certificate>

<eid2:X509Certificate>MIIGKDCCBBCgAwIBAgIBATANBgkqhkiG9w…</eid2:X509Certificate>
    </eid2:SignatureCertificateChain>
  </dss:OptionalOutputs>
  <dss:SignatureObject>
    <dss:Other>
      <!-- [Mandatory] The SignTasks element (namespace EID 2.0 DSS EXTENSIONS)
provides output data from the signature generation process. -->
      <!-- Note that this element is marked as mandatory, but that of course only
holds if the service operation succeeded. -->
      <eid2:SignTasks xmlns:eid2="http://id.elegnamnden.se/csig/1.0/dss-ext/ns">
        <eid2:SignTaskData SignTaskId="SignTask1" SigType="XML" AdESType="None">
          <eid2:ToBeSignedBytes>AAECAwQFBgcICQoLDA0ODw==</eid2:ToBeSignedBytes>
          <eid2:Base64Signature Type="http://www.w3.org/2001/04/xmldsig-more#ecdsa-
sha256">9Ozy8x2gsbDlbdVLR9aC3Cf9v3VW4jDhGswR1wJ+jcxnNWFLgYpgZv/23uP8Gt93tM/bI65NV7k
38SFSosMN1g==</eid2:Base64Signature>
        </eid2:SignTaskData>
        <eid2:SignTaskData SignTaskId="SignTask2" SigType="PDF" AdESType="None">
          <eid2:ToBeSignedBytes>AAECAwQFBgcICQoLDA0ODw==</eid2:ToBeSignedBytes>
          <eid2:Base64Signature Type="http://www.w3.org/2001/04/xmldsig-more#ecdsa-
sha256">9Ozy8x2gsbDlbdVLR9aC3Cf9v3VW4jDhGswR1wJ+jcxnNWFLgYpgZv/23uP8Gt93tM/bI65NV7k
38SFSosMN1g==</eid2:Base64Signature>
        </eid2:SignTaskData>
      </eid2:SignTasks>
    </dss:Other>
  </dss:SignatureObject>
</dss:SignResponse>
</SignResponse>
```

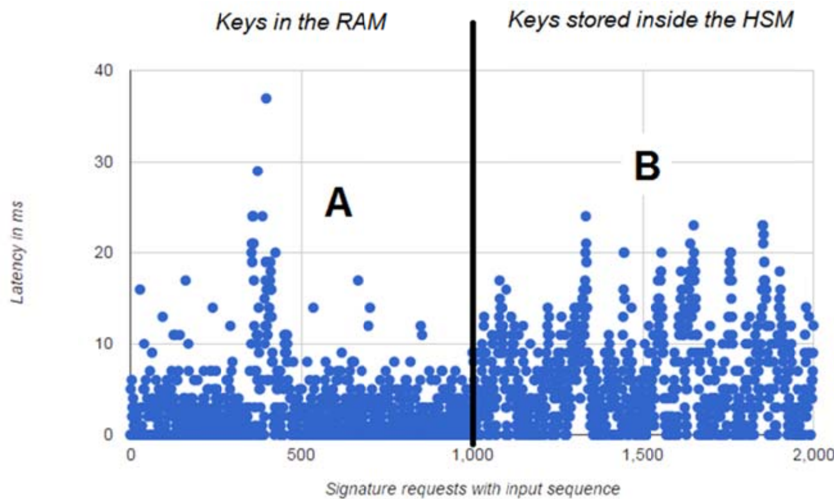
Appendix B: Test results

The following three tests were performed to check the behavior of the HSM and signing service when key pairs come from four different places: HSM's RAM, stored keys inside the HSM, Key Pool, and key generation. Since we did not have access to whole storage of the HSM and to simplifying the test we assumed that HSM RAM has the capacity of 1000 instead of 12033 key pairs. In the same way we assumed that the storage capacity is 1000 rather than 20000 key pairs. Two tests were performed with two different key pool sizes 1000 and 3000 key pairs.

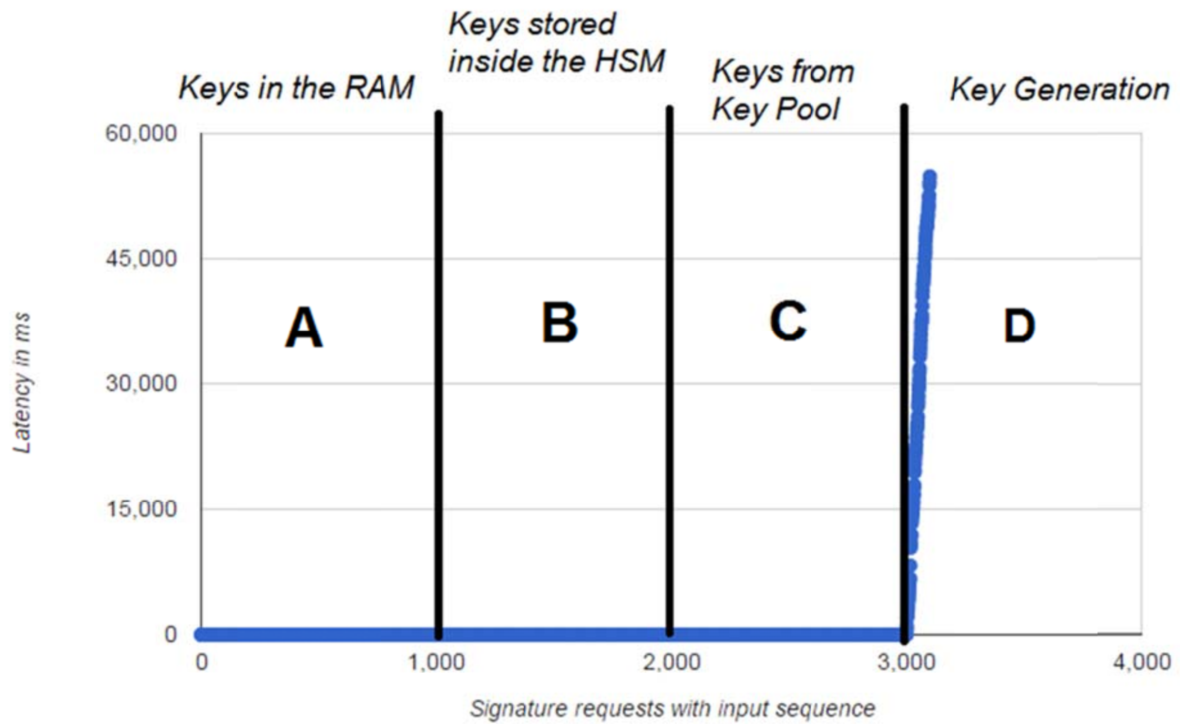
In each test two graphs show the same test results. The first includes the key generation part with high latency time and the second excludes the key generation time to be able to show the other parts with greater detail. The corresponding portions of each chart are marked with the same capital characters.

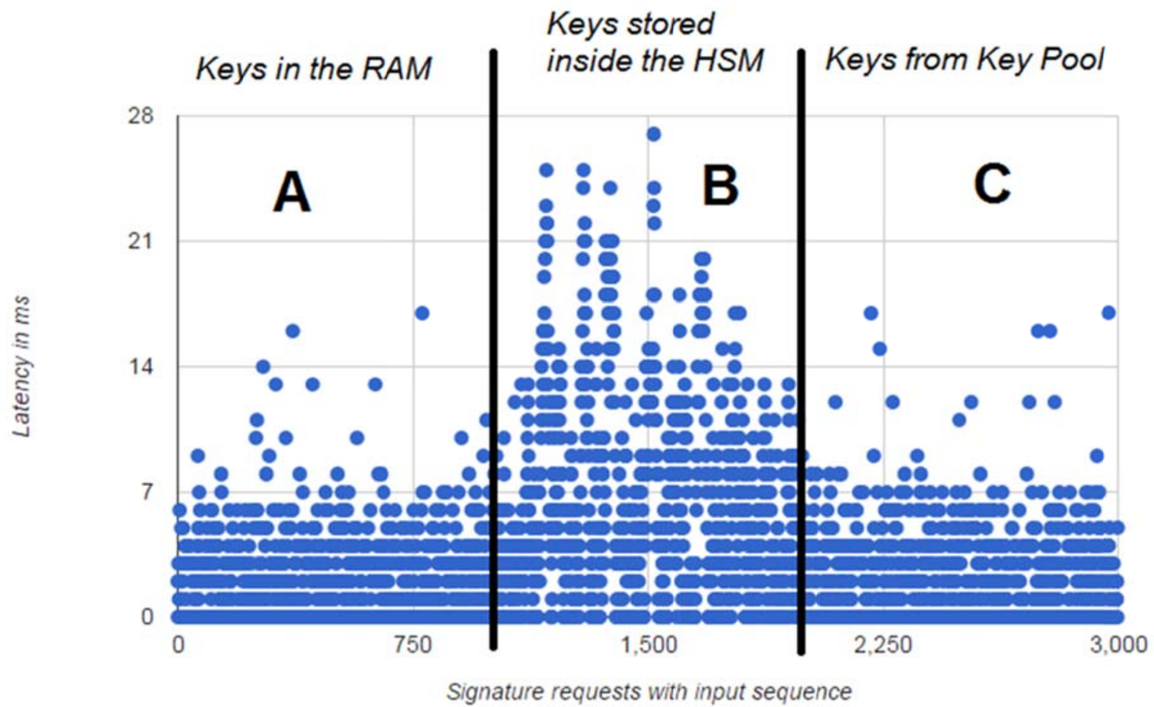
Test 1 : 1000 sign with Keys in the RAM (**A**) , 1000 sign with keys in the HSM's storage (**B**) and 100 sign with key generation (**C**)



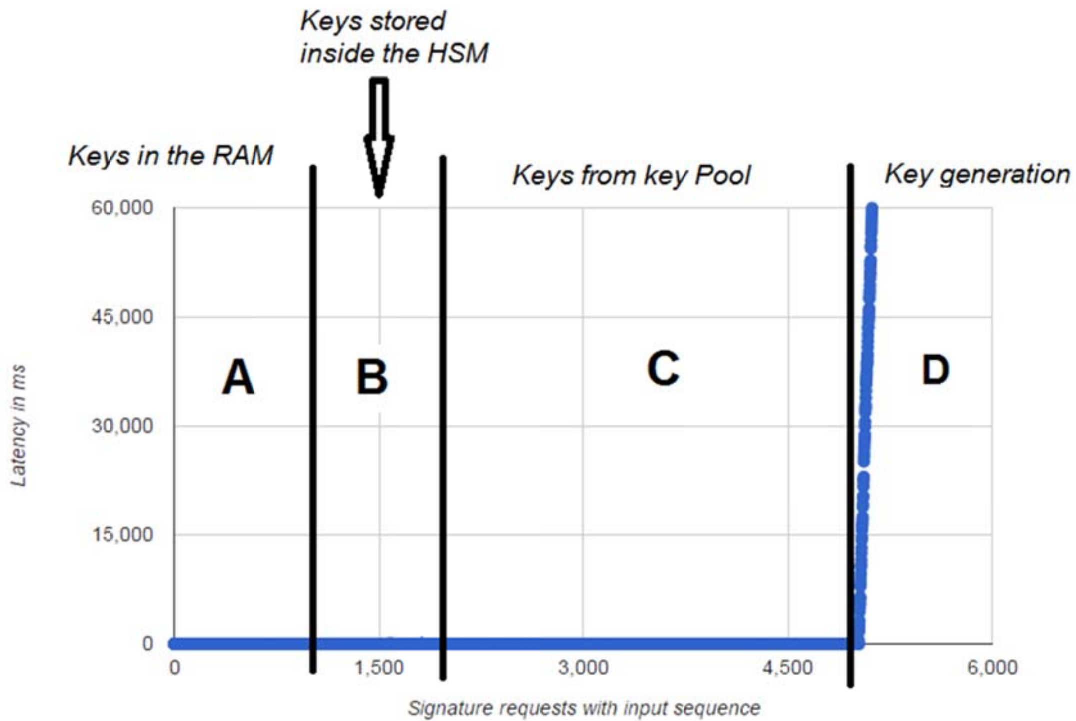


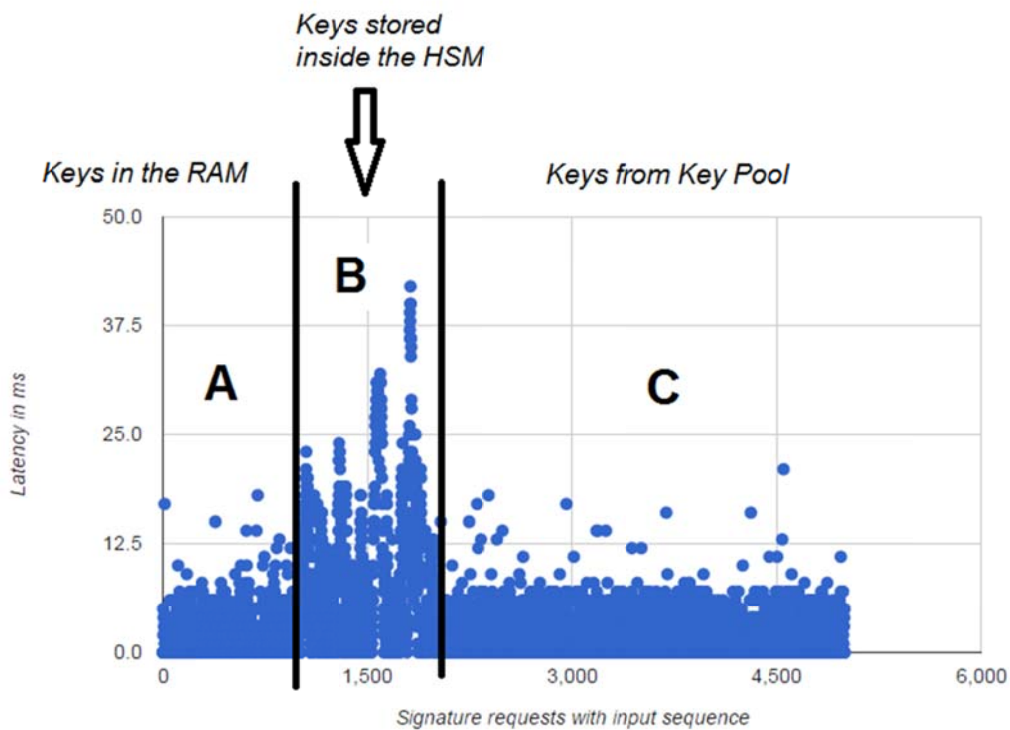
Test 2 : 1000 sign with Keys in the RAM (**A**) , 1000 sign with keys in the HSM's storage (**B**), 1000 sign with Key Pool (**C**) and 100 sign with key generation (**D**)





Test 3 : 1000 sign with Keys in the RAM (**A**) , 1000 sign with keys in the HSM's storage (**B**), 3000 sign with Key Pool (**C**) and 100 sign with key generation (**D**)





Appendix C: Luna SA 1700 HSM Performance report (internal by SafeNet)

RSA										
Operation	Key Size	Data Size (Bytes)	PKCS#11				Java			
			Single HSM		2xHA		Single HSM		2xHA	
			Ops/Sec	Latency (msec)	Ops/Sec	Latency (msec)	Ops/Sec	Latency (msec)	Ops/Sec	Latency (msec)
Sign	1024	16	1800	2,1	3600	2,3	1800	2,1	3600	2,2
	2048	16	350	9,1	700	9,4	350	9,1	700	9,2
	4096	16	50	60	98	60	50	59	97	60
	8192	16	3,6	280	3,6	280	3,6	280	3,6	280
Verify	1024	16	5300	0,68	9600	0,84	5300	0,74	9800	0,84
	2048	16	4300	0,94	8300	1,1	4400	0,99	8600	1,1
	4096	16	2800	1,5	5500	1,6	2800	1,6	5500	1,6
	8192	16	97	10	94	11	97	10	96	10
Hash/Sign	SHA256 with RSA-1024	1k	1700	2,3	3400	2,5	1800	2,2	3600	2,3
	SHA256 with RSA-2048	1k	350	9,3	660	9,5	350	9,2	700	9,3
	SHA256 with RSA-4096	1k	50	59	98	60	49	60	98	60
	SHA256 with RSA-8192	1k	3,6	280	3,6	280	3,6	280	3,6	280
KeyGen	1024	-	11	98	3,6	280	13	76	4,3	210
	2048	-	1,8	590	1,3	790	1,3	560	1,1	650
	4096	-	0,17	15000	0,067	10000	0,17	6000	0,13	7500
	8192	-	0,008	120000	0,025	40000	0,017	60000	0,008	120000
Encrypt	1024	16	5800	0,69	10000	0,85	5400	0,72	10200	0,84
	2048	16	4600	0,94	8900	1,1	4700	0,99	9200	1,1
	4096	16	2900	1,5	5700	1,6	3000	1,5	5800	1,6
	8192	16	110	9,3	100	9,6	110	9,3	110	9,4
Decrypt	1024	16	1800	2,1	3600	2,2	1800	2,2	3600	2,2
	2048	16	350	9,1	700	9,4	350	9,1	710	9,3
	4096	16	49	59	100	60	49	59	98	59
	8192	16	4,6	220	4,7	210	4,6	220	4,6	220

TRITA-ICT-EX-2014:182

www.kth.se