Intrusion Fault-Tolerance using Threshold Cryptography

Final Report

May 2nd , 2004

Abhilasha Bhargav, Rahim Sewani and Sarvjeet Singh
Department of Computer Science
Purdue University, West Lafayette IN

Abstract

There is a need of guaranteeing the authenticity of messages sent by a group of individuals to another group. Despite the increasing awareness of security, malicious failures are inevitable in the modern world. We had proposed to develop a fault-tolerant group communication system using threshold cryptography to exchange messages. Due of unreliability of a single entity in an electronic environment we want to distribute the trust model over a group of entities. We are assuming the Byzantine fault model for the system where no one is trusted. Our contribution this semester is the RSA threshold signature library, and its analysis.

1 Introduction

With the increasing dependence of the society in the modern computing systems availability and trustworthiness become very important. Traditional unicast communication models and protocols like Transmission Control Protocol (TCP) support point–to–point communication and were designed for applications involving communication between no more than two processes at a time, usually a client and a server. However, many modern applications for example an online game being played by several participants around the world, or a multimedia conference where participants communicate on a shared white board, involve more than two users exchanging information. Hence, this requires multi–point–to–multi–point communication.

Group communication provides this multi–point–to–multi–point communication by organizing processes in groups. Groups can be seen as dynamic sets of entities where users can choose when they wish to join or leave a group. Depending on how messages are exchanged, the literature distinguishes two different types of group communication namely one-to-multiple communication and multiple-to-multiple communication [1].

We started our project looking into the Spread toolkit which provides group communication services for wide area networks. One important function of the Spread is keeping a consistent view of the system for successful group communication. It achieves this by having the servers exchange messages. Here an adversary can disrupt the view by sending incorrect messages. Therefore there is a need for authenticated group communication in Spread.

The servers communicate in their own subnet by sending broadcasts. Therefore within a subnet, servers can trust each other since they all get the same messages. However, such trust does not exist between subnets and mechanisms have to be implemented for it. One way would be to get a signed message from each server of the subnet from where the message is coming. This is very expensive and therefore not realistic. The other way would be to trust a threshold of servers in the other subnet. To achieve this we proposed integrating threshold signature library into the servers.

Threshold signature schemes enable a group of n entities share a private signature key in such a way that, for some parameter k ($1 \leq k \leq n$), any subset of k entities can

collectively create a valid signature on a message, whereas any collection of k - 1 or fewer entities cannot. To the best of our knowledge there is no open source Threshold Signature toolkit available. Therefore we implemented a library for RSA Threshold Signatures as described in Practical Threshold signatures by Shoup [8].

This report is organized as follows- first we describe the Threshold signatures: the general concept, highlights of Shoup's implementation and related work. Then we give a detailed description of the RSA Threshold Crypto Library we implemented including the API and the system requirements. This is followed by detailed analysis of the Threshold RSA with respect to the traditional RSA scheme with individual signatures. This includes both theoretical and experimental analysis. Finally for threshold signatures section we describe its general advantages and disadvantages. We then briefly explain the Spread architecture and implementation suggestions and analysis. That is followed by our accomplishments, future work and conclusion.

2 Threshold Signatures

2.1 General Concept

Threshold cryptography allows a party of say n people to share the ability of performing a cryptographic operation (e.g. creating a digital signature). Any t parties (t < n) can perform this operation jointly. This t is called the threshold. It is infeasible for any t-1 parties (or less) to do so, even by collusion. Also, the secret cannot be recovered by any subset of parties.

Different methods of sharing secrets have been studied. One of them is the Lagrange Interpolating Polynomial Scheme which is described as follows:

A polynomial of degree t-1 is determined by it values at t distinct values of its argument. In numerical analysis it is shown that given t points $(i_1,k_1),\ldots,(i_t,k_{it})$ with different x coordinates i_j ,there is a unique polynomial of degree ≤ t-1 passing through them. The Lagrange polynomial is as follows:

$$h(x) = \sum_{s=1}^{t} K_{is} \prod_{\substack{j=1 \\ j \neq s}}^{t} (x-i_j)/(i_s-i_j)$$

The different shares come from a random polynomial of degree t-1:

$$h(x) = (a_{t-1}x^{t-1} + \ldots + a_1x + a_0) \bmod p$$

with the term a_0 equal to the secret key and the rest of the coefficients being random. Then the different shares can be calculated as

$K_i = h(i)$ for $1 \leq i \leq w$. Given these shares the Lagrange polynomial can be reconstructed and any t-1 shares cannot do so [12].

2.2 Shoup's Protocol

We implemented the Shoup's Protocol 2 described in [8] because it is the first practical scheme for implementing threshold signatures. It uses the RSA signature scheme in a very efficient manner. The scheme uses only one level of secret sharing, to sign a message, each server simply sends a single response to a signature request and must do work that is equivalent up to a constant factor to computing a single RSA signature. No further interaction is needed to recover from faults.

It assumes a trusted dealer to generate keys. This is caused by the fact that it relies on a special property for the RSA modulus, namely it must be the product of two safe primes. There is recent work [11] to implement the schema without the trusted dealer. They showed using some non standard assumptions that the proof of correctness described in the paper can be carried out without the strong primes. Nevertheless there are other papers [14, 15] which mention that while the requirement for safe primes can sometimes be avoided this typically comes at the cost of extra communication, computation, and or nonstandard intractability assumptions. We strictly implemented the protocol 2 in the Shoup paper because it was more computationally efficient than protocol 1 in the same paper [8].

2.3 Related work

The Shoup's protocol has the following properties: Firstly it is unforgeable and robust in the random oracle model, assuming the RSA problem is hard. Second, signature share generation and verification is completely non interactive, and finally the size of an individual signature share is bounded by a constant times the size of the RSA modulus.

A lot of related research work has used the Shoup protocol to make it distributed and schemes which join techniques combining undetachable RSA threshold signatures with it [16]. The value of such method when applied to the mobile agent scenario has also been described in [17].

3 RSA Threshold Crypto Library

3.1 API of the Library

The top level main functions of the library and their descriptions are as follows:

```
int TC_Combine_Sigs(TC_IND_SIG** ind_sigs,
TC_IND *key, BIGNUM *hM, TC_SIG *sig);
/*
TC_Combine_Sigs takes the individual signatures ind_sigs from the group members and tries to generate a valid signature on the message hash hM. key is either the individual key (generated by TC_get_ind or the combine key (generated by TC_get_combine). The final signature (if success) is returned in sig. ind_sigs array is generated by making one or more calls to set_tc_sig.

Return value is TC_NOERROR in case of success, TC_ALLOC_ERROR in case allocation of temporary BIGNUM's failed, TC_BN_ARTH_ERROR in case these was an error while doing arthmetic on BIGNUM's, and TC_NOT_ENOUGH_SIGS if ind_sigs does not contain enough ( >= threshold) of verified signatures. In all the library functions that returns eitherTC_ALLOC_ERROR or TC_BN_ARTH_ERROR, the error codes and texual messages can be obtained by using openssl > crypto > err(3)
 */

TC_DEALER* TC_generate(int bits, int l, int
k);
/*
TC_generate sets up the threshold signature system by generating the public keys, private keys and the verification keys. These individual keys could be extracted from the TC_DEALER struct by making calls to TC_get_ind (individual key), TC_get_combine(combine key), TC_get_pub (public key). The hash function used for verication can be set by TC_Dealer_setHash (A EVP_md5() has is assumed if no hash function is specified). The public key generated will be of 2*bits bits (typically bits=128). l is total number of members and k is the threshold. The pseudo random number generater should be seeded using RAND_seed or RAND_add (see openssl > crypto > rand(3) for details) before making call to TC_generate. The TC_DEALER struct should be freed using TC_DEALER_free() to prevent memory leaks.

Return value is TC_NOERROR in case of success, TC_ALLOC_ERROR in case allocation of temporary BIGNUM's failed and TC_BN_ARTH_ERROR in case these was an error while doing arthmetic on BIGNUM's. In all the library functions that returns either TC_ALLOC_ERROR or TC_BN_ARTH_ERROR, the error codes and texual messages can be obtained by using openssl > crypto > err(3)
 */

int TC_Check_Proof(TC_IND *tcind,BIGNUM*
hM,TC_IND_SIG* sign, int signum);
/*
TC_Check_Proof checks the proof of correctness sent by individual member. tcind is either the secret individual key or the combine key, hM is hash of the message (which was signed), sign is the signature sent by the member, and signum is the member's number who generated "sign" (the same number that was used to extract member's key from TC_DEALER). This function is called by the TC_Combine_Sigs internally to check all the signatures and may not be normally used. But in cases, when you want to detect "mallicious" group members, you can use this function to check their proof.

Return value is 0 if the proof is not valid, 1 if its valid, TC_ALLOC_ERROR in case allocation of temporary BIGNUM's failed and TC_BN_ARTH_ERROR in case these was an error while doing arthmetic on BIGNUM's. In all the library functions that returns either
```

```
TC_ALLOC_ERROR or TC_BN_ARTH_ERROR, the error codes and texual messages can be obtained by using openssl > crypto > err(3)
 */

int genIndSig(TC_IND *tcind,BIGNUM *hM,TC_IND_SIG* sign);
 /*
genIndSig generates a signature and proof of correctness on the message hash hM, using tcind as the secret key. The output is placed in sign. sign should already be allocated using TC_IND_SIG_new() before passing it to genIndSig. After its used, it can be freed by TC_IND_SIG_free to prevent memory leaks. The pseudo random number generater should be seeded using RAND_seed or RAND_add (see openssl > crypto > rand(3) for details) before making call to this function.

Return value is TC_NOERROR in case of success, TC_ALLOC_ERROR in case allocation of temporary BIGNUM's failed and TC_BN_ARTH_ERROR in case these was an error while doing arthmetic on BIGNUM's.
 */


int TC_verify(BIGNUM *hM, TC_SIG sig, TC_PK *tcpk);
/*
TC_verify verifies the signature sig (generated using TC_Combine_Sigs) on the message hM using the public key tcpk.

Return value is 0 if the signature is not valid, 1 if its valid, TC_ALLOC_ERROR in case allocation of temporary BIGNUM's failed and TC_BN_ARTH_ERROR in case these was an error while doing arthmetic on BIGNUM's.
 */
```

The helper functions are described below:

```
/* Helper functions on structs defined. */

TC_DEALER *TC_DEALER_new(void);
 /* Allocates and returns a new TC_DEALER struct. Returns NULL on error */

void TC_DEALER_free(TC_DEALER *tc);
 /* Frees the TC_DEALER struct generated by TC_DEALER_new or TC_generate */

void TC_DEALER_print(TC_DEALER *tc);
 /* Prints TC_DEALER struct */

int TC_Dealer_setHash (TC_DEALER *tcd,unsigned short hashpointer);
 /* Changes the default hash function (EVP_md5()) used for verification by tcd */

TC_IND *TC_get_ind(int index,TC_DEALER *tcd);
 /* Extracts the index member's private key from tcd. Should be freed using TC_IND_free to prevent memory leak */

TC_PK *TC_get_pub (TC_DEALER *tcd);
 /* Extracts the public key from tcd. Should be freed using TC_PK_free to prevent memory leaks */

TC_IND *TC_get_combine(TC_DEALER *tcd);
 /* Extracts the combine key from tcd. This key is similar the ind key except that it doesn't have the secret share and hence can only be used for combining and not signing the individual messages. Should be freed using TC_IND_free to prevent memory leaks
 */

void TC_PK_free(TC_PK *tcpk);
 /* Frees memory used by tcpk */

void TC_IND_free(TC_IND *tcind);
 /* Frees tcind. tcind can be a secret/private key or a combine key */

TC_IND_SIG *TC_IND_SIG_new();
 /* Allocates and returns a new TC_IND_SIG struct. Should be freed using TC_IND_SIG_free */

void TC_PK_Print(TC_PK *pk);
 /* Prints pk */

void TC_IND_Print(TC_IND *ind);
 /* Prints ind */

void TC_IND_SIG_Print(TC_IND_SIG *sig);
 /* Prints sig */

TC_IND_SIG **TC_SIG_Array_new(int l);
 /* Returns a new array of size l(=total number of members), which is to be used as target of one or more set_TC_sig calls to set the individual signatures, and finally it is passed to
```

```
TC_Combine to combine all the individual signatures
 */

void set_TC_SIG(int index, TC_IND_SIG* si,TC_IND_SIG** sigs);
 /* Adds the individual signature si to index slot of sigs. Index should be number of member whose signature is si. sigs should be allocated using TC_SIG_Array_new before making this call. After sigs is used in TC_Combine, it should be freed using TC_SIG_ARRAY_free to prevent memory leaks
*/

void TC_IND_SIG_free(TC_IND_SIG *a);
 /* Frees TC_IND_SIG a */

void TC_SIG_Array_free(TC_IND_SIG **a,int l);
 /* Frees TC_IND_SIG **a. l is total number of members */
```

Finally the Marshal and Demarshal functions to convert the TC's structs into stream of bytes and convert the streams of bytes back into the structs. This is useful if you want to transfer the structs from one machine/process to other. The stream of bytes produces is independent of the platform (big endian, small endian) and thus can be used to transfer these structures across platforms. All structs returned by demarshal should be freed by corresponding calls to free().

```
*//* Marshal, Demarshal fucntions */

int TC_PK_size(TC_PK *a);
 /* Returns size of byte stream produced by marshaling a */
int TC_PK_marshal(TC_PK *a, unsigned char *buf);
 /* marshals a and stores it in buf. buf should be atleast TC_PK_size(a) long.
Returns TC_PK_size(a) in case of success, -1 on error
 */

TC_PK *TC_PK_demarshal(unsigned char *buf);
 /* Demarshals buf and allocates and returns back a TC_PK structure.
Returns the struct in case of success, NULL on error (e.g. cannot parse buf, cannot allocate TC_PK).
 */

int TC_IND_size(TC_IND *a);
 /* Returns size of byte stream produced by marshaling a */

int TC_IND_marshal(TC_IND *a, unsigned char *buf);
 /* marshals a and stores it in buf. buf should be atleast TC_IND_size(a) long.
Returns TC_IND_size(a) in case of success, -1 on error
 */

TC_IND *TC_IND_demarshal(unsigned char *buf);
 /* Demarshals buf and allocates and returns back a TC_IND structure.
Returns the struct in case of success, NULL on error (e.g. cannot parse buf, cannot allocate TC_IND).
 */

int TC_IND_SIG_size(TC_IND_SIG *a);
/* Returns size of byte stream produced by marshaling a */

int TC_IND_SIG_marshal(TC_IND_SIG *a, unsigned char *buf);
 /* marshals a and stores it in buf. buf should be atleast TC_IND_SIG_size(a) long.
Returns TC_IND_SIG_size(a) in case of success, -1 on error
 */

TC_IND_SIG *TC_IND_SIG_demarshal(unsigned char *buf);
 /* Demarshals buf and allocates and returns back a TC_IND_SIG structure.
Returns the struct in case of success, NULL on error (e.g. cannot parse buf, cannot allocate TC_IND_SIG).
 */

int TC_SIG_size(TC_SIG a);
/* Returns size of byte stream produced by marshaling a */

int TC_SIG_marshal(TC_SIG a, unsigned char *buf);
 /* marshals a and stores it in buf. buf should be atleast TC_SIG_size(a) long.
```

```
Returns TC_SIG_size(a) in case of success, -1 on error
 */
TC_SIG TC_SIG_demarshal(unsigned char *buf);
/* Demarshals buf and allocates and returns back a TC_SIG structure.
Returns the struct in case of success, NULL on error (e.g. cannot parse buf, cannot allocate TC_SIG).
 */
```

3.2 System Requirements

The library has been made as generic as possible although the OpenSSL Crypto library is needed to be installed in the machine.

4 Analysis

4.1 Theoretical Analysis of Threshold RSA signature compared to single RSA signatures

The asymptotic analysis is given in the table below where k is the threshold. We refer to our implementation of the RSA Threshold Signatures as TC-RSA and the traditional RSA implemented in the OpenSSL Crypto library as RSA.

	TC-RSA	RSA
Size of Signature	O(1)	O(k)
Generate Individual Signature	O(k)	O(k)
Merging Signatures	O(k)	N/A
Signature Verification	O(1)	O(k)

The size of the signature in TC-RSA is simply O(1) because all the partial signatures are combined to one as compared to traditional RSA where all the k signatures are required. Then to generate individual signatures the time is bounded by the threshold of signatures. The same bound holds true for combining of the TC-RSA signature. There is no combining of signatures in the single RSA schema. Finally in the verification since there is only one TC-RSA signature, O(1) time is required whereas the RSA would need verification of all k signatures. Note that in RSA all the different public keys have to be matched with the signer whereas in TC-RSA there is only one public key for the group of signers.

4.2 Experimental Analysis of Threshold RSA signature compared to single RSA signatures

We compared our Threshold Signature library with the OpenSSL RSA signatures. The total number of members participating (n) was taken as 100 and the threshold (k) was 67. Since we are considering the Byzantine fault model we do need 2/3rd's of the group to be honest.

The time was measured in micro-seconds using gettimeofday() library call before and after the different functions. The machine used was a Dell Workstation with P4 1.8 GHz processor, 256 MB of RAM and Linux OS. The four functions calculated were 1) Dealer 2) Individual signature generator 3) Combiner and 4) Verifier. They are described as follows:

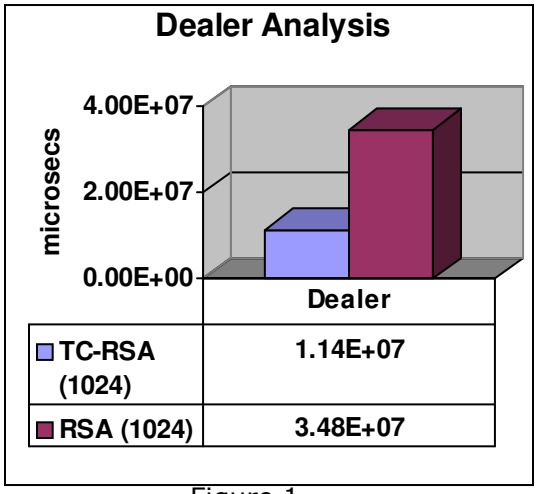

Figure 1

The dealer in RSA has to create 100 public, private key pairs and does considerably more work than the TC-RSA which creates 100 shares of the secret and one public key.

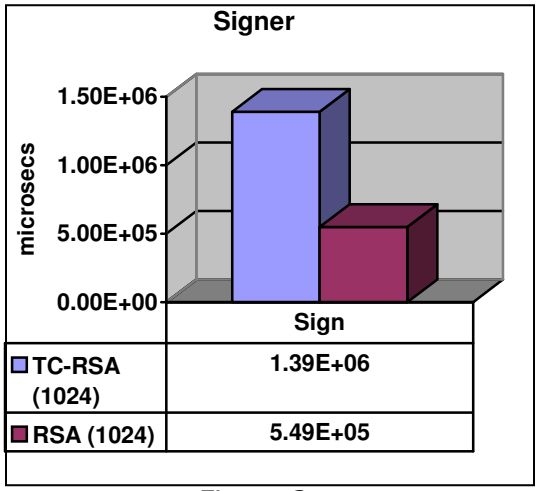

Figure 2

The signing in the TC-RSA takes more time because of the need of the proof of correctness (‘z’ , ‘c’) as described in the Shoup’s paper [8].

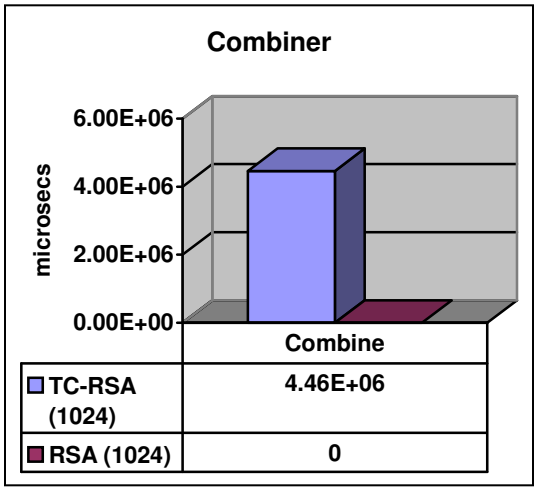

Figure 3

There is no combining of the individual RSA signatures and the 67 (threshold) signatures take 4.46 seconds to combine. 67 is relatively large number as compared to the thresholds we will have for the Spread toolkit.

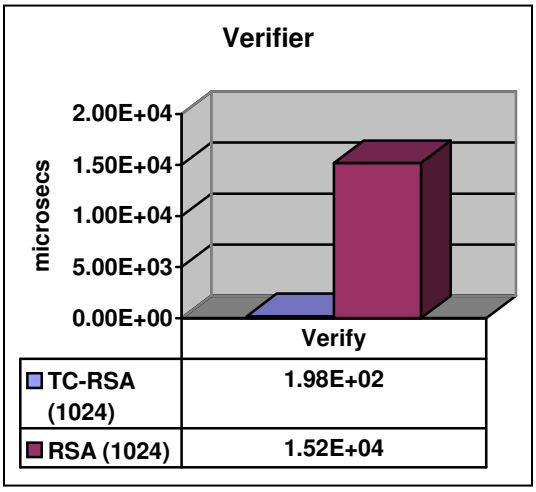

Figure 4

The verification holds one of the most important advantages of using the TC-RSA as compared to traditional individual signatures. As we would expect the RSA takes time in the order of approximately 70 times of the time taken for the one TC-RSA signature.

4.3 Setup cost analysis

Typical RSA requires PKI and the verification of n signatures would require the person verifying to know n public keys where as in TC-RSA one needs to know only one public key, hence the setup cost is minimal.

4.4 Dealer Risk Analysis

In the case of TC-RSA only one dealer needs to be compromised as compared to RSA where k out of n dealers needs to be compromised to generate k invalid sets of secret keys resulting in invalid signatures. Here the RSA seems to be more secure.

4.5 Hop by hop Combining vs. Single Combiner

We think that the TC-RSA protocol can be modified in a way such that the combining is done hop by hop. In TC-RSA a malicious combiner cannot remove a particular signature from the combined set of signatures. It can still remove the entire signature. In traditional RSA and adversary can remove any particular signature since they are just appended with each other and can be distinguished.

4.6 Threat Analysis of TC-RSA compared to RSA

TC-RSA requires 2 secure hash functions whereas RSA requires one. Individual signature phase and verifier threats are the same.

5 Advantages of Threshold Signatures

RSA Threshold signature has several advantages. It is secure and robust in the random oracle model assuming the RSA problem is hard. The signature share generation and verification are completely non- interactive. Also the size of an individual signature share is bounded by constant times the size of the RSA modulus.

It is easy to change the secret shares without changing the secret itself i.e. by having a new polynomial with the same free term. It allows for and hierarchical scheme in which the number of pieces needed to determine secret depends on their importance. The combiner can verify the individual signatures before combing the whole message. The combiner of the signature cannot determine which of the k shareholders created the signature. The combiner does not recover the secret key and thus cannot forge the group signature. By choosing different k and n it provides tradeoff between security and reliability.

6 Disadvantages of Threshold Signatures

One disadvantage of threshold signatures include is the necessity of the dealer, an entity who computes the shares and distributes them, is trusted. We also need an entity that combines all the pieces in order to recover the key and both can be central points of trust and failure.

7 Spread

7.1 Introduction

Spread is a toolkit that provides a high performance messaging service that is resilient to faults across external or internal networks. Spread functions as a unified message bus for distributed applications, and provides highly tuned application-level multicast and group communication support. Spread services range from reliable message passing to fully ordered messages with delivery guarantees, even in case of computer failures and network partitions. [1]

7.2 Spread Architecture

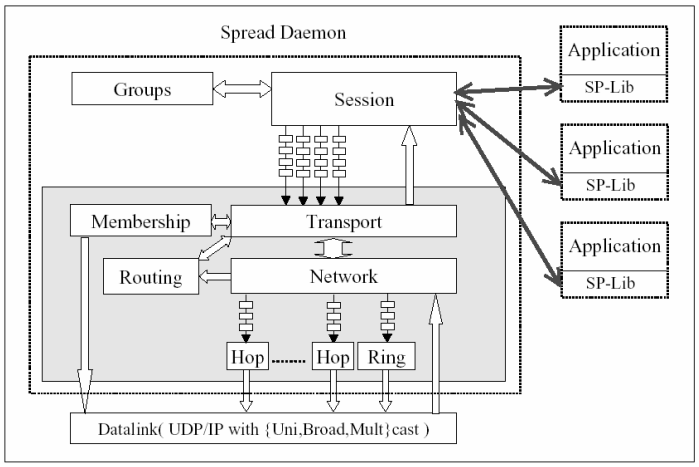

Figure 5 Spread Architecture

Spread architecture, as shown in Figure 5, integrates two low-level protocols: one for local LANs called Ring, and one for WANs connecting them, called Hop. Based on this architecture, Spread decouples the dissemination and local reliability mechanisms from global ordering and stability protocols. This allows many optimizations useful for WAN settings.

Spread is a very flexible group communication service suitable for LANs as well as for WANs. It provides priority channels to the application and open–group semantics where a sender does not have to be a member of the group in order to multicast to it. Spread, written in ANSI C, is available for all common platforms and different client APIs (C, C++, Java) are supported. Another advantage of Spread is that these APIs are very simple and their use is independent from the configuration. Thus, the same application can be used in LANs as well as in WANs. The performance of Spread depends on the configuration of the system [1].

7.3 Implementation suggestion of TC-RSA in Spread

In previous work [13] distinction was made among two basic approaches to integrate security services into client-server group communication systems. The layered

architecture which places security services in a client library layered atop the GCS client library and the integrated architecture which has the security services at the servers. We planned to implement the Threshold Signature toolkit in the latter architecture. Understanding and implementing the threshold signature toolkit is a part of the future work for this project.

7.4 Analysis

For the setup of the Spread servers the configuration file should contain all the potential servers. The dealer phase should be completed before setting up the configure files for the servers. The secret key, verification keys for its own subnet and the public keys for the other subnets should be set in this file. The dealer could be any secure machine dedicated for this purpose.

In the case of network partition when each partition has at least the threshold number of servers then there is no problem. Both these partitions continue to work similar to normal Spread as they both can generate valid group signature on their messages. If any partition has less than the threshold servers then the partition can not produce a valid signature. In this case, it cannot communicate with rest of Spread servers and will be cut off from it until it can communicate with at least threshold of its members.

The key distribution scheme described above suggests static keys with fixed threshold. For security reasons, it would be sometimes desirable to refresh the secret keys of a group. While this is possible, it introduces need of a trusted party (dealer) to refresh the keys. Thus, we discourage key refresh and any change in the value of the threshold because any group of malicious servers can claim a network partition and establish a new threshold. The key refresh is possible only if we keep the dealer alive continuously. This would be more vulnerable because it has a single point of failure and introduce many complexities because of the security problems.

8 Accomplishments

During the first few weeks of our project we did extensive research and understood the current state of work in group communication protocols and threshold cryptography. Then we studied Spread architecture and the interaction between its modules. We spent majority of our time implementing and testing the RSA Threshold Signature library successfully. Although the protocol was described in the Shoup paper[8] we had to understand and solve several intricate details including the detailed mathematical functions and then its implementation using the OpenSSL crypto library. We also did its threat and complexity analysis which will be useful for those implementing our library.

9 Future Work

Our threshold signature library can be now integrated in the spread communication system followed by its testing and performance analysis.

10 Conclusion

Although the motivation to implement the RSA threshold signature library was motivated by its use in implementing authenticated group communication in the Spread toolkit, we believe that this library can be used in a very generic fashion. We have it open source and have tested it thoroughly and has no known bugs. Its analysis with single RSA also gives the tradeoffs of using such a scheme instead of traditional individual RSA signatures.

11 References

[1] Secure Group Communication Systems Master's Thesis of Science by Gernot Schmölzer (March 2003) www.iaik.tu-graz.ac.at/teaching/11_diplomarbeiten/archive/Schmoelzer.pdf

[2]Admission Control Admission Control in Peer Groups in Peer Groups http://sconce.ics.uci.edu/docs/gac_seminar.pdf

[3] D.Dolev. The Byzantine Generals Strike Again

[4] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems, 4(3):382-401, July 1982.

[5] M.Castro, B.Liskov. Practical Byzantine Fault Tolerance

[6] S.Mishra, N.P.Subraveti, S.Tanaraksiritavorn. Issues in Building Intrusion Tolerant Group Membership Protocols, Shivakant Mishra

[7] Group Communication Specifications: A Comprehensive Study by Gregory V. Chockler, Idit Keidar and Roman Vitenberg http://www.ee.technion.ac.il/people/idish/ftp/gcs-survey.pdf http://www.ee.technion.ac.il/people/idish/ftp/GroupCommunication.pdf

[8] Practical Threshold Signatures, Victor Shoup http://shoup.net/papers/thsig.pdf

[9] Spread Toolkit Documentation http://www.spread.org/docs/docspread.html

[10] OpenSSL Project http://www.openssl.org/

[11] Practical Threshold RSA Signatures Without a Trusted Dealer, by Ivan B. Damgard and Maciej Koprowsk www.brics.dk/RS/00/30/BRICS-RS-00-30.pdf

[12] Cryptanalysis of Number Theoretic Ciphers by Samuel S. Wagstaff

[13] Scaling Secure Group Communication Systems: Beyond Peer-to-Peer.Yair Amir, Cristina Nita-Rotaru,Jonathan Stanton, Gene Tsudik http://www.cnds.jhu.edu/pub/papers/discex3_scalable.pdf

[14] E_cient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products by Algesheimer et. al. (2002) http://www.zurich.ibm.com/security/publications/2002/alcash02.pdf

[15] Fully Distributed Threshold RSA under Standard Assumptions by Fouque, Stern http://citeseer.ist.psu.edu/556204.html

[16]Undetachable Hreshold Signatures by Chris et. al. http://citeseer.ist.psu.edu/539180.html

[17] On the value of threshold signatures Niklas Borselius et. al. http://citeseer.ist.psu.edu/532301.html