UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

PKI and high-level security programming abstractions for SNOOP

Håkon Stenhaug

INF-3981 Master's Thesis in Computer Science - December 2019

Abstract

When working with sensitive data in a distributed setting, secure multi-party computations(SMC) protocols aims to preserve the privacy of participants. A core aspect of any SMC protocol is secure communication between participants. Based on the NOOP and SNOOP middleware with a combination of secret key and public-key cryptography we design and implement a public key infrastructure(PKI) to share signed certificates containing a participants credentials and public key. Components of the PKI support signing and verification of certificates, public-key encryption supports signing and verification of communication between participants. Our tools provide high-level security programming abstractions to support the development of SMC protocols.

Contents

Abstract **i**

List of Figures **iv**

List of Tables **v**

List of Code Examples **vi**

1 Introduction **1**

2 Related work **3**

2.1 Secure multi-party computations . . . 3
2.2 Public-key Infrastructure . . . 4
2.2.1 x509 Certificate . . . 5
2.2.2 PEM . . . 6
2.3 NOOP . . . 6
2.4 SNOOP . . . 8

3 Design & Implementation **9**

3.1 Simple PKI . . . 9
3.2 Certificate . . . 10
3.3 Message . . . 12
3.4 Blob . . . 13
3.4.1 File I/O . . . 14
3.4.2 Encrypt & Decrypt . . . 15
3.5 Communication . . . 15
3.5.1 Socket . . . 16
3.5.2 Server . . . 16
3.6 Node . . . 17
3.6.1 Message handling . . . 18
3.6.2 Sharing certificate . . . 19
3.6.3 Certificate from sign request . . . 19
3.6.4 Sharing blob . . . 20
3.7 Test applications . . . 20
3.7.1 Sharing data between nodes . . . 20

3.7.2 Secure multi-party computation 21

4 Evaluation **23**
4.1 Performance 23
4.1.1 TCPServer 23
4.1.2 Blob 25

5 Discussion **27**
5.1 Optimization 28
5.1.1 Abstractions 28
5.1.2 Encryption operations 29
5.1.3 Socket 30
5.2 PKI 30
5.3 Security & Privacy 31
5.4 Support for SMC 32
5.5 SMC Protocols 33

6 Conclusion **35**

List of Figures

2.1 Public key infrastructure model 4
2.2 Hierarchy of trust in a PKI 5

3.1 Simple public key infrastructure model 10
3.2 CA and participant nodes in ideal model 22

4.1 Blob operation performance with varying chunk size 26

5.1 Web of trust 31

List of Tables

2.1 NOOP extended types. 7

3.1 Notation 9
3.2 Message types 18
3.3 Operations for get certificate. 19
3.4 Operations for post certificate. 19
3.5 Operations for get certificate sign request 20
3.6 Operations for post blob. 20

List of Code Examples

2.1 Timing execution of code with NOOP 7
2.2 NOOP tcpsend and tcpreceive 7
2.3 NOOP cryptography 8
3.1 RSA keys 11
3.2 Certificate generation 11
3.3 Certificate bytes 11
3.4 Certificate public key 12
3.5 Message usage 13
3.6 Blob partial data 14
3.7 Blob usage 15
3.8 TCPsocket usage 16
3.9 TCPserver usage 16
3.10 TCPserver loop 17
3.11 Nodes 18
3.12 Share data multiple nodes 21
3.13 Setup certificates 21
3.14 Trusted third party 22
4.1 PKI test 24
4.2 Timed execution of Blob operations 25
5.1 Abstraction of share operation 28
5.2 Abstraction of receive blobs 29
5.3 Secret key bundling for multiple recipients of same Blob 29

Chapter 1

Introduction

Patient related health data are typically located at different general practices and hospitals. When processing and analyzing such data, the provided infrastructure and toolset has to take into consideration legal, security and privacy issues. In the SNOOP[2] project the combination of secure multi-party computations (SMC) algorithms, encryption, public key infrastructure (PKI), certificates, and a certificate authority (CA) is used to implement an infrastructure and a toolset for statistical analysis of health data. In a previous project[4] we implemented a set of crypto-primitives in SNOOP (capstone project autumn 2018). SNOOP is an adaptive middleware for secure multi-party computations (SMC). SNOOP is an extension of the adaptable component based middleware NOOP[1]. It combines support for secure multi-party computations, encryption, public key infrastructure (PKI), certificates, and certificate authorities (CA). SNOOP and the deployment of SNOOP applications have to take into consideration legal, security and privacy issues involved in statistical analysis of such data. SNOOP tries to support a wide range of possible SMC algorithms and computing graphs. It provides high-level programming abstractions that adapt to the current run-time environment at deploy time. Contracts are provided to match the application requirements with available run-time functionality and requirements. Current SNOOP (and NOOP) prototype is implemented in Python.

In this project we will focus on high level security programming abstractions based on the crypto primitives, a more robust implementation of the primitives, and the development of a simple PKI for experimentation with the high level security programming abstractions. We will demonstrate how these programming abstraction can be used to describe and implement different security protocols and systems with high level of abstractions.The project includes analysis and decisions on available crypto primitives, programming abstractions, message formats, file formats. During the first half of the projects additional functionalities will be considered. This includes timestamps, message numbering, generation and use of nonce, implementation of high-level programming abstractions for digital signing and verification and a set of test applications.

Chapter 2

Related work

2.1 SECURE MULTI-PARTY COMPUTATIONS

The problem introduced by Yao[15] was stated as "Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?" Suppose you want to compute the value of a function $f(x_1, x_2, ...x_n)$ where participant p_i only knows the value of x_i and no other x. How do you compute the value of f without learning the value of any one x. In the case of the millionaire the number of participants n = 2, the computation would output who was richer and not the combined wealth as you could easily reveal the other participants value. This is the problem of secure multi-party computations (SMC).

SMCs involves multiple participants (nodes) jointly performing some computation. Each participant contributes with its private input to the computation, but the private input must remain hidden from other parties or any third party. Participants an SMC may lack trust between each other. The SMC algorithm must preserve the privacy of all parties involved. SMC studies introduce the concepts of *ideal* and *real* models[16]. In an ideal model the parties of an SMC are joined by a trusted third party who receives input from each participant through a secure channel, computes the function and shares the output with each participant. In a real model participants are mutually untrusted, there exists no trusted third party and the participants compute the function among themselves. A protocol in the real model aims to simulate the environment of the ideal model, where no participant receive any input of other participants. A real model protocol is said to be secure if no participant can learn about other participants more than it could in an ideal model.

Security must be preserved even with adversarial behavior by one or many participants. An adversary can be semi-honest or malicious. A semi-honest

adversary is a participant who follows the protocol properly, but provides false input or attempts to learn about other participants input in some way. A malicious adversary will may attempt to disrupt the execution of an SMC. Not following protocol, false input, not starting the execution, suspending execution or aborting at any time. Adversarys may collude to learn about honest participants.

With security in the context of SMC we mean security of communication, that any data we want hidden is inaccessible by any eavesdroppers. Privacy in an SMC is the privacy of a participants input, an honest participant should have the privacy of their input preserved. Privacy is preserved by SMC protocols.

2.2 PUBLIC-KEY INFRASTRUCTURE

A PKI is a trusted third party with a set of components that binds public keys to an entity through the use of digital certificates. The purpose of the PKI is to provide services for identification, verification and authentication of users and their public keys. A user of a PKI is not necessarily a person, but can be any entity, e.g a server, application, component of a system, etc. We will mainly refer to users as entity. The main components of a PKI are the certificate authority and registration authority (RA). The PKI enables sharing and authentication of public keys through certificates to preserve privacy of exchanged data between users. A digital certificate is digital document containing an entities credentials and public key. The registration authority is a management entity where a node

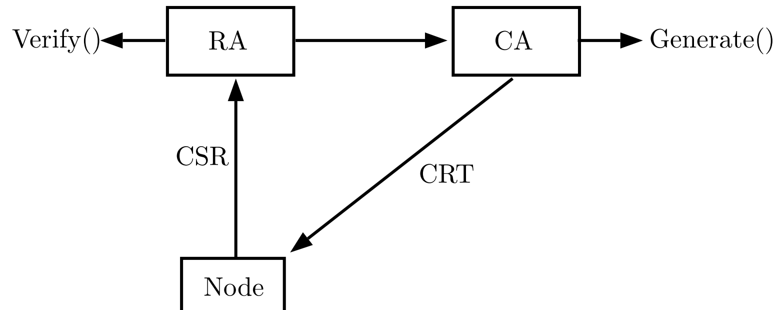


Figure 2.1: Public key infrastructure model

can make itself known to the CA. The RA manages nodes, verification of credentials, and certificate sign requests. Verified requests are forwarded to a CA who generates and issues a signed certificate. The responsibility of a CA is to be a trusted node that generates certificates signed by its own private key, which nodes can use to enable trust among each other through verifying the CA's signature with the CA's own certificate. The CA maintains a list of certificates available so other nodes can get them. A revocation list may also be used if a

node is compromised and can not be trusted before a certificate expiration is due. The CA stores each issued certificate to allow nodes to request another nodes certificate issued by the CA. Figure(2.1) shows the components of a PKI with the steps of a node acquiring a certificate from a CA.

PKI is a hierarchical centralized model of trust. The root CA certificate (highest level trusted entity) is self signed.Otherwise a CAs certificate is issued and signed by a higher level CA in the PKI. If we trust a CA, we also trust any higher level CA in the PKI. Figure(2.2) shows a hierarchy of trusted entitys in a PKI, the number specifies the trust level with 0 being the most trusted(root) CA.

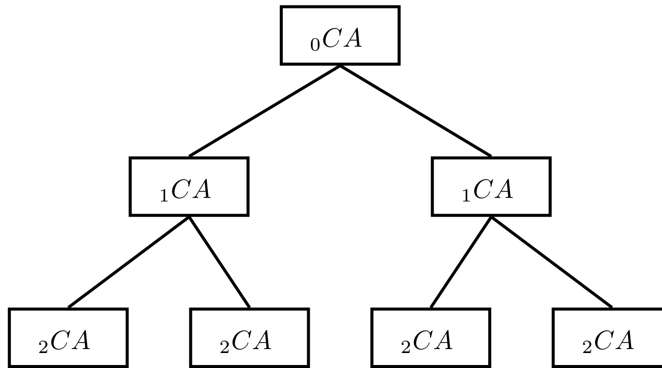


Figure 2.2: Hierarchy of trust in a PKI

2.2.1 X509 CERTIFICATE

x.509 (v3) is a format standard for digital certificates described in RFC5280[5]. A certificate contains information about the identity of the owner(subject) and identity of entity who generated and signed it (issuer). The issuer signs the certificate with their own private key. Components defined in the x.509 specifications are certificate authority (CA), registration authority (RA), end entity (node, user/subject of certificates), certificate revocation list (CRL) issuer, and repository (system that stores certificates and CRLs, and distributes these to end entities). The purpose of the certificate is best described by directly quoting the specifications[5].

> “Users of a public key require confidence that the associated private key is owned by the correct remote subject (person or system) with which an encryption or digital signature mechanism will be used. This confidence is obtained through the use of public key certificates, which are data structures that bind public key values to subjects. The binding is asserted by having a trusted CA digitally sign each

> certificate. The CA may base this assertion upon technical means (a.k.a., proof of possession through a challenge- response protocol), presentation of the private key, or on an assertion by the subject. A certificate has a limited valid lifetime, which is indicated in its signed contents. Because a certificate's signature and timeliness can be independently checked by a certificate-using client, certificates can be distributed via untrusted communications and server systems, and can be cached in unsecured storage in certificate-using systems."

Certificates are used in a wide range of applications to enable trust and secure communication. In this thesis we follow the x509 specification for certificates and a hierarchical public-key infrastructure model.

2.2.2 PEM

Privacy Enhanced Mail (PEM)[8] is a base64 encoded file format used for x509 files. The format is based on a set of standards defined in [9]. The file is prefixed with a "–BEGIN ... —" and ends with a " –END ... —" line. In NOOP, asymmetric keys and certificates are encoded as PEM to be stored on disk or represented in bytes format. When later referring to bytes representation or serialized keys or certificates, the objects are encoded in the PEM format and given as bytes.

2.3 NOOP

NOOP is an experimental platform for developing distributed applications. To introduce type safety, interfaces, and a component model in Python, NOOP introduces a type language and a way to apply typing to functions (and methods). This type system is used to create interfaces and a software component model for distributed applications. NOOP is designed to provide adaptive high-level programming abstractions support the development of complex distributed applications. These high-level programming abstractions include support for a set of security functions based on symmetric and public-key crypto systems.

NOOP provides features for type checking through the Signature decorator, periodically running a method through the Activity module, communication through sockets with tcp and ip modules, and crypto operations with asymmetric, symmetric and certificate modues. The signature decorator allows us to type safe method arguments, return values and exceptions. Along with type checking NOOP introduces the types constructors opt, one and whatever. These are used for the signature decorator to extend python types.

NOOP has a *Timer* module for timing the execution of a piece of code. When the execution is finished the timer prints the result to a given output (defaults to standard output i.e command line). In a previous project [4] we extended the module to allow for multiple iterations to display not just a single time of execution, but number of timed exectutions, average and total execution

Table 2.1: NOOP extended types.

opt(t),	optional value, of type t or no value.
one(t_1, t_2)	one value of type t_1, t_2, ..., or t_n.
whatever	a value of any type.

Listing 2.1: Timing execution of code with NOOP

```
timer = Timer()
with timer("foo␣bar")
        foo_bar()
timer.finish()
```

time. This was added with storing each timer call and a finish method to finalize the timing. Listing(2.1) shows example usage of the timer module. The code in the with statement is timed, the string argument given is a name to identify what is being timed and accessing the timed data. The identifier ("foo bar" in example) allows us to time multiple different executions with unique identifiers before calling finish.

Socket communication is implemented in NOOP as the *tcp* and *ip* modules. Ip provides an IPaddr class for managing host name, port, and other socket arguments. While tcp implements the socket operations for accepting, connecting, sending and receiving data built on the information provided by IPaddr. Combined these allow for minimal socket management by the user. The tcp module provides functions tcpsend and tcpreceive, these are called with an IPaddr object and will setup the socket connection (or reuse if already opened). Example of these functions are shown in Listing (2.2). Both functions sets the address as the servers (host, port) pair. Data is sent as bytes, therefor must be encoded before sending.

NOOP adds crypto operations built on the cryptography python package[7]. The implementation provides abstractions for easy to use crypto operations, with default values to avoid detailed options such as key size and public ex-

Listing 2.2: NOOP tcpsend and tcpreceive

```
Server side:
        a = IPaddr(node=<host>, port=4576)
        msg = tcpreceive(a).decode("utf-8")
        tcpsend(a, ("Received:␣" + msg).encode("utf-8"))

Client side:
        a = IPaddr(node=<host>, port=4576)
        tcpsend(a, "Hello␣world!".encode("utf-8"))
        msg = tcpreceive(a).decode("utf-8")
```

Listing 2.3: NOOP cryptography

```
New key:
        privkey = RSAPrivKey()
        pubkey = privkey.getpubkey()
From disk:
        privkey = RSAPrivKey(pem=keybytes)
        pubkey = privkey.getpubkey()

Key operations:
        cipher = pubkey.encrypt("hello world".encode())
        plain = privkey.decrypt(cipher)

        sig = privkey.sign(cipher)
        if pubkey.verify(sig):
                print("verified!")
```

ponent for keys, and padding schemes for key operations (encrypt, decrypt, sign, verify). For symmetric encryption Advanced Encryption Standard (AES) is implemented, and for asymmetric (or public-key) NOOP implements RSA. These keys can be converted to a bytes format for storing on disk or for sharing. Keys can be initialized from an existing key in bytes format or if no argument given a new key is generated. We can also create an empty key class with a nokey boolean flag when initializing. Data must be given in bytes format before encrypting or decrypting.

2.4 SNOOP

SNOOP is an adaptive middleware for secure multi-party computations (SMC). It combines support for secure multi-party computations, encryption, public key infrastructure(PKI), certificates, and certificate authorities (CA). It is used to perform statistical analysis of electronic health record(EHR) data. SNOOP is a middleware built to support the constructions, deployment and execution of applications performing statistical analysis of EHR data. It is an extension of NOOP.

In SNOOP the distribution of work in an SMC is managed by a coordinator node and a set of sub-processes. The coordinator generates a computing graph for the SMC algorithm and each node in the graph is a sub-process. Each nodes have an address and a unique public/private key pair.

Chapter 3

Design & Implementation

We design and implement a simple public key infrastructure (PKI) for secure communication and sharing of data between nodes. The core functionality of a PKI is generating and issuing trusted certificates. To do this we need secret and public-key crypto functionality, node to node communication, and bytes data management. This is realized by expanding the NOOP certificate module, implementing a message format, and data management modules.

Table 3.1: Notation

$\{m\}$	some bytes m
$\{m_n\}$	some bytes m belonging to node n
$_s\{m\}$	encrypt m with secret key(aes) s
$\{m\}_p$	encrypt m with public key(rsa) p
$\{m\}^n$	m signed by node n
$\{m\}_p^n$	m signed by node n, encrypted by p
$A->B:\{m\}$	send message containing m from A to B

3.1 SIMPLE PKI

We implement a simple PKI for managing and generating certificates, the CA absorbs the responsibility of the RA. We do not require heavy authentication in the context of our use of a PKI and we focus on the generation of certificates and verification of the trusted CA's signature. To implement the PKI we expand on NOOPs crypto modules for certificates. We require the use of certificate sign requests and certificates to share public keys. These will be shared between nodes and must be serializable and deserializable. For our version of a simple PKI we want to focus on session based verification of key ownership, certificates that are valid for a short time. In our context a session is an execution of a SMC protocol. We require that a node can act as a trusted CA through generating

and signing certificates for other nodes, so that we can verify the certificate was issued by the trusted CA using the CA's own self signed certificate.

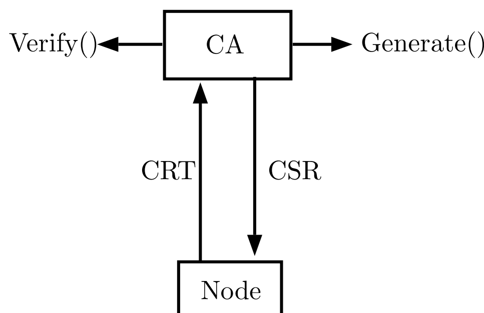


Figure 3.1: Simple public key infrastructure model

The CA must do some verification on a nodes credentials, we are not focusing on the authentication part of a PKI so this is handled by verifying correct message format and signed messages by a node. Communication is implemented using NOOPs tcp and ip modules for socket operations and messages formatted as JSON strings. Nodes communicating with each other will share their certificates, which they verify with the CAs certificate. Any message with a signature not matching the public key from their certificate issued by the CA is discarded. A node in our PKI is launched with a set address for the CA, this means that any nodes communicating must have the same node set as a CA.

3.2 CERTIFICATE

We use certificates to share public keys and related information such as various attributes (name, address, etc) about owner (subject) and node that generated the certificate (issuer), expiration date, and a signature from the issuer private key. This signature allows us to trust a certificate from an otherwise untrusted node, by verifying the signature with the public key from the trusted nodes certificate (which is usually self signed).

NOOP supports cryptography operations and encryption keys built on the python cryptography[7] package. To implement certificate features we build on existing NOOP modules, and add certificates as a module in the NOOP package. The module implements x509 digital certificates for RSA keys only. The module features certificate sign request (CSR) and certificate (CRT) objects, and methods for generating, representing the objects as bytes, and verifying signatures.

Both CSR and CRT inherit a base class for shared functionality and static variables, and are initialized with the same parameters. All paremeters are optional, however methods may fail unless certain variables are set. CSR requres subject name attributes while CRT additionally require issuer name attributes.

CSR is generated with a private key, and CRT is generated with a public key. Parameters include key, subject, issuer and pem. If pem is given as parameter the other fields are irrelevant as they will be loaded from the pem bytes.

Subject and issuer contains name attributes and is an object of the class x509.name from the cryptography[7] package. The certificate module includes a helper function set_x509_name to simplify generating these name attributes. Attributes include:

```
COUNTRY_NAME,
STATE_OR_PROVINCE_NAME,
LOCALITY_NAME,
ORGANIZATION_NAME,
COMMON_NAME.
```

This helper function has default values set if no parameters are passed or if a default parameter is set to True. The attributes tells us information about the certificate subject and issuer. COMMON_NAME represents the host name of the certificate owner and should match the host address, e.g "localhost.com". For the CSR class key is of type RSAPrivateKey, and only subject is necessary to generate the CSR. For Cert, key is of type RSAPublicKey, both subject and issuer are necessary to generate the certificate. To generate the certificate a RSAPrivateKey needs to be given, this key should belong to the issuer. The key given for a CSR is a private key as the CSR is self signed when generated and the associated public key stored in the CSR. Unlike a certificate where the private key signing does not need to be associated with the certificates public key.
To use the certificate module, we first need an RSA keypair.

Listing 3.1: RSA keys

```
privkey = RSAPrivKey()
pubkey = privkey.getpubkey()
```

We can then set up and generate the certificate or csr object.

Listing 3.2: Certificate generation

```
csr = CSR(subject=subject, key=privkey)
csr.generate()

crt = Cert(subject=subject, issuer=issuer, key=pubkey)
crt.generate(privatekey)
```

Listing(3.3) shows how to get bytes representation of the objects, and how to load the objects from bytes. These methods are necessary when sharing CRT and CSR data with other nodes.

Listing 3.3: Certificate bytes

```
csrbytes = bytes(csr)
crtbytes = bytes(crt)
csr = CSR(pem = csrbytes)
crt = Cert(pem = certbytes)
```

Public keys stored in a certificate can be accessed from the key variable or getkey method. The key variable holds an RSAPrivateKey object from the cryptography package and getkey method returns a NOOP RSAPrivKey object. Verifying a certificates signature is done by giving a RSA private key. This separation of key object types is a design choice as the certificate object makes no use of the NOOP RSA key object methods, but when extracting the public key from the certificate we want to perform encrypt and verify operations from the NOOP implemtation. The general use is therefor the getkey method when extracting a public key from a certificate. Example in Listing(3.4) shows using keys and verifying a signature using a CA certificate public key.

Listing 3.4: Certificate public key

```
pubkey = CAcrt.key
pubkey = CAcrt.getkey()
if crt.verify(pubkey)
        print "verified!"
```

3.3 MESSAGE

The message module provides methods for building, loading, signing and verifying messages. A message is built as a JSON formatted string. One of the goals when designing a message format, was for it to be to easy expand on. A JSON formatted string is easy to read for humans, which is useful for debugging, it is easy to generate and parse, and it is simple to add new fields to the message.

Below is a sample message showing the various fields used and their expected data type.

```
msg = {
        "message": {
                "meta":
                        {
                        "method":str ,
                        "args":str ,
                        "id":str ,
                        "message_id":int ,
                        "timestamp":str ,
                        "payload_type":str(type(payload)) ,
                        "extra":whatever
                        },
                "payload":one(str , bytes)
                }
        "signature":str
        }
```

A message consists of two primary fields meta and payload. The payload can be of any type support by python JSON, if the type is byte the payload must be encoded to work with JSON. In this implementation we convert the bytes to a hex string. The payload type is added to the meta field so the recipient knows how to handle the payload, and which conversion (if any) is required.

Meta fields consists of method, args, id, message_id, and extras. Timestamp is an automatic field set when the message is built, it is a string generated from date and time. Method is a string identifier for the message type, an example message type in our PKI implementation for requesting another nodes certificate is "get/crt". Args is a string of additional optional arguments. There are two fields for identification, one for the message sender and one for the message itself. Message sender means the node that generates the message, and will be the nodes id. Message id is optional and can be used if message numbering is required. Extras is a field to put any other additional metadata, the type must be compatible with JSON. Extra meta data can for example be information about file contents when sharing a file, such as size and name.

The message string is signed when building the message if a private key is given, and the signature is encoded and added to the message (see above example format). If no key is given the signature field will be empty. We sign the entire message string to ensure validity of the message. To accomplish this we must first build the message JSON string, encode it as bytes to sign it, and build a new JSON string with the message and signature.
Listing(3.5) is an example of how we use the module when building and loading a message.

Listing 3.5: Message usage

```
Building a message
        m = Message(key=privatekey, identifier="node 1")
        msg = m.build("post", "hello world")

Loading a message
        m = Message()
        payload = m.load(msg)
        if m.verify(pubkey) == True
                do something
```

As can be seen above, loading a message returns the payload. All base fields can be reached from same name variables in the message object, such as $m.id$, $m.args$, $m.extra$ and so on. When verifying the signed message we need to ensure the string we verify is identical to the string signed. As our message is built with a nested JSON string, loading requires two JSON load calls. One for the full message and one for the nested message containing meta and payload. We found the multiple load calls necessary as the message field was represented as a string rather than a dictionary when loading once.

3.4 BLOB

Blob is a bytes data management module designed to simplify encrypting and decrypting any size data. Data can exist on disk or in memory, and we want blob operations to work seamlessly regardless of size of data or where it exists. The module features operations for storing, loading, encrypting and decrypting bytes. In a blob, data is stored in memory or on disk depending on initial

parameters and size of data. A blob object points to this data, with a variable(in memory) or a file path(on disk), operations on the blob does work on the data the blob points to. We treat all data pointed to as bytes. A blob can be initialized with bytes data, path, and a secret key (AES), as well as boolean variables *ondisk* and *clean*. If ondisk is true, we store data on disk (this is true by default), and if false we keep it in memory. If clean is set true, we remove old files after encrypt and decrypt operations (default is false). If no path is given and we store data on disk, a random name is generated where we will store the data. A user of a blob is responsible for management and any file stored on disk (even from random generated name) will exist until a blob delete method is called.

Blob.set() is a method used to update the blob content and info, where all parameters are optional. Any excluded parameters are ignored in the update. This method is called when initializing the object and after encrypt and decrypt operations. It can also be manually called to set update info, set a new key, or even reuse the object for new data. If path is given we point the blob to the new path and generate any directories if necessary, if data is given we overwrite the data of the blob. Size of data the blob points to and a exists tag is also set. Blob operations can not function if data does not exist. Set can also be used to update the blobs secret key, with either the key bytes or an instance of the NOOP AESKey class.

3.4.1 FILE I/O

Blob has methods for loading and storing data, which can also be done on a partial pieces of the data with store_part and load_part methods. Working on parts of data are useful when dealing with large files where we can not hold it all in memory at once or when we cant work on too large data, such as when encrypting and decrypting, or sending bytes to another node. We load or store a part by the parameters *pos* or n, and *size*. *pos* is the specific byte position to start from, n is the n'th chunk of bytes in the data, *size* is the size in bytes to store or load. The size of a chunk is determined by a blob variable *chunksize* if the size parameter is not given. All parameters for loading and storing are optional, but one of *pos* or n is required for it to work.
The example in Listing(3.6) shows two ways of working on parts of blob data. Both examples copies blob data back to the same file on disk, essentially doing nothing.

Listing 3.6: Blob partial data

```
for n in range(blob.chunks):
        part = blob.load_part(n=n)
        blob.store_part(data, n=n)
for pos in range(0, blob.size, chunksize)
        part = blob.load_part(pos = pos, size=chunksize)
        blob.store_part(data, pos = pos)
```

3.4.2 ENCRYPT & DECRYPT

Encrypt and Decrypt methods were adapted from a previous capstone project[4] where we implemented a model for end-to-end encryption of cloud storage data. We encrypt or decrypt data by loading a set limit of data at a time defined by a *chunksize* variable, and append the result to an output. For files the output is a same name file with an added (encrypt) or removed (decrypt) ".AES" extension, for data in memory we discard the input data with the output.
Examples:

Listing 3.7: Blob usage

```
Data on disk
        blob = Blob(path="foobar.txt", clean=True)
        blob.encrypt()
        blob.decrypt()

Data in memory
        blob = Blob(data=b"hello world", ondisk=False)
        blob.encrypt()
        cipher = blob.load()
        blob.decrypt()
        plain = blob.load()
```

When a blob is encrypted or decrypted, the blob points to the new data. For crypto operations we can give an optional output path. If a file already exists on the output file path a number is added next to the file name, e.g "lorem.txt" becomes "lorem(1).txt"

When a blobs use if expended, we can call a delete method. This will remove the file from disk and empty the blobs path to clean blob variables. We have a flag *clean*, where if false will not remove the file from disk when delete is called. This is useful when iterating a list of blobs but some might be files you do not want removed, such as blobs initialized from disk file paths, while blobs initialized from bytes with a random generated name stored on disk should most likely be removed.

3.5 COMMUNICATION

NOOP implements a tcp socket module, this module is however limited to a single socket connection per host/port pair and does not support concurrent sockets. While this is sufficient in many cases for communication, we want the ability to concurrently receive messages from multiple nodes. This has been implemented to work with our node message handling. We want an address connecting to a remote address to function as before, only one socket handle is required on an outgoing connection. For incoming connections we want to handle all connections even if some are still open. We also want to ensure that any code previously written using TCPsocket module will not be affected by the changes.

3.5.1 SOCKET

NOOP TCPsocket implements bind, listen and accept as a single accept method that closes the listening socket after a connection has been accepted. To handle concurrent connections we re implement the accept method to not close listening socket after each accepting a connection. Accept will attempt to accept the first valid connection similar to the old implementation and return an instance of a new TCPcon class. This class contains an id and a socket handle for receiving or sending.

Listing 3.8: TCPsocket usage

```
Old:
        s = TCPsocket(addr)
        s.accept()
        m = s.receive()
        s.send("hello world".encode())
New:
        s = TCPsocket(addr)
        con = s.accept()
        m = con.receive()
        con.send("hello world".encode())
```

3.5.2 SERVER

We also implement a TCPserver class using the updated TCPsocket with TCP-con. The server is started in a thread, and each accepted connection is threaded for concurrency. The accepted connection is passed to a thread to receive the initial message, before being forwarded to a handler method. The handler is called with the received message and TCPcon given as an arguments. When the handler is complete we close the socket and remove the TCPcon. A server is launched with an IPaddr and a handler method, with an optional int n for number of connections accepted before shutting down. Server stop can be called before the number of accepts have been reached.

Listing 3.9: TCPserver usage

```
server = TCPserver(add=adr, handler=handler_method)
server.start(n=10)
server.stop()
```

The server thread can be stuck on waiting to accept a connection. If a call to stop the server is made the server will wait for one more connection to be established before shutting down. To handle this we implement a timeout. If a connection is not established within a timeout t, the server loop continues and if a shutdown is called we break the loop and join the remaining handler threads.

The sample code in Listing(3.10) shows how a server thread operates to handle concurrent connections and a timeout for graceful shutdown of the server. A timeout float value is set and if a connection is not established in this time we jump to the next iteration of the loop. When we finish using a node, we can call a clean method for shutting down the server, removing certificates and blobs.

Listing 3.10: TCPserver loop

```
s = TCPsocket(addr)
self.running = True
i=0
while self.running:
        if i>=self.n and self.n != 0:
                break
        try:
                con = s.accept(timeout=1.0)
        except TimeoutError:
                continue
        t = threading.Thread(target=self._receive, args=(
            con))
        t.start()
        i = i + 1
```

This will call a server shutdown on the node, a delete on the nodes tcpsocket to close all open sockets, and a delete on all blob objects for certificates and blobs. Python garbage collection should take care of closing remaining sockets, but we still do it when cleaning the node in case of errors.

3.6 NODE

A node is a sub-process that is part of a network with a unique private/public key pair and certificate. The module provides node to node communication methods for securely sharing certificates and data. The features provided by the Node are implemented using the NOOPs crypto and tcp/ip modules, and the message and blob modules described above. It is designed with the intention of designing specific node types extending the functionality, such as handlers for more messages, and doing work on received blob data. The node does not keep track of other nodes itself.

Each node has a unique identifier. This is used to name data about the node, such as certificates. We get the identifier by hashing the string representation of the nodes ip/port pair, this has the benefit of checking for an existing certificate before an operation. As those attributes are unique to a node, it ensures the identifier is also unique. The identifier is included in every message and allows recipient node to look up the certificate if it exists locally, then do some encryption or verification of signature.

A node has a host name, port, RSA keypair, an optional certificate, optional path, a list of blobs for this node, and a list of certificates(as blobs) shared with this node. A certificate is required to communicate with other nodes, apart from receiving their certificates. Path is the nodes root where it stores any blobs, this root path is set as the nodes identifier if not given. The node message server can be run for a set number of messages received, if not given will run until a stop is called.

Listing(3.11) is an example of two nodes where one requests the certificate of another, the server expects one message and will stop after it is received.

Listing 3.11: Nodes

```
Listening node
        node = Node(host="localhost", port=4576)
        node.server_run(n=1)

Requesting node
        node = Node(host="localhost", port=4578)
        adr = IPaddr(node="localhost", port=4576)
        node.get_certificate(adr)
```

The module is designed with basic features that any entity of a multi-party computation requires with the intention of expanding additional features for specifics node types, such as CA or RA nodes, or the coordinator node from [2]. Additional features may include certificate revocation lists and certificate sign request authentication of the subjects credentials. In our base node a certificate sign request is accepted if the request message is signed with the private key associated with the attached public key.

3.6.1 MESSAGE HANDLING

Table 3.2: Message types

get/crt	Get certificate from node.
get/csr	Get a new certificate generated and signed by node.
post/crt	Post own certificate to node.
post/blob	Post a blob to node.

When receiving a message a handler is called by removing non a-Z characters from the message method field, and calling a function node.handle_method. for example "get/crt" handler would be node.handle_getcrt. A message handler takes the message string as input, aswell as the socket handle for the connection, the handle is required as some methods use multiple messages to and from a node to complete. The node method on the sender side is implemented with try, except, finally. We first try to establish a connection, build and send the message, then receive response (if any, or multiple), if any exception is raised we abort execution. In the finally clause we clean up and close the connection to recipient node. Default message types handled by the base Node are shown in Table(3.2).

3.6.2 SHARING CERTIFICATE

The first step to securely share a blob, is to exchange node public keys. Public keys are store in certificates, so we send a request for the nodes certificate. We use the message module to build a request. To share certificate with a node we send a post/crt. To get a nodes certificate we send a get/crt, the response message will then contain the nodes certificate.

Get certificate can also function as a certificate exchange. The get/crt message payload can be empty or contain a certificate, the message signature is verified with the attached certificate. This saves us from building, sending and handling an additional message for posting a nodes certificate when preparing for securely sharing data. For clarity we separate the exchange into post and get certificate when describing operations.

Table 3.3: Operations for get certificate.

$n_1 -> n_2 : \{m\}^{n_1}$	message m signed by n
$n_2 -> n_1 : \{r, crt_2\}^{n_2}$	response containing certificate signed by n

Table 3.4: Operations for post certificate.

$n_1 -> n_2 : \{crt_{n_2}\}^{n_1}$	m signed by n

Requests for a certificate in a nodes list of certificates can be sent if the receiving node is acting as a CA. This is similar to getting the nodes certificate except the message contains args for node id. The handler will look up the certificate with that node id in their list of certificates and return that instead of their own. If the certificate is not found we return an error response instead.

3.6.3 CERTIFICATE FROM SIGN REQUEST

If node n_1 wants a certificate signed by a another node n_2, usually a CA, we use the message method "get/csr". Node n_1 generates a certificate sign request containing subject name attributes about n_1 and a public key . Node n_2 will then generate a certificate based on this info and sign it with their own private key. This means that any node can function as a certificate authority, along with the issuing nodes certificate we can verify the certificate was generated by a specific node or CA.

Table(3.5) shows the messages for sending a certificate sign request from node n_1 and generating a certificate signed by node n_2.

Table 3.5: Operations for get certificate sign request

$n_1 - > n_2 : \{m, csr_{n_1}\}^{n_1}$	send sign request from node 1 to node 2
$n_2 - > n_1 : \{m, crt_{n_1}^{n_2}\}^{n_2}$	return message with node 1 certificate signed by node 2

3.6.4 SHARING BLOB

When sharing a blob we first send share request which contains information about the blob, such as encryption key, blob name, and blob size. The secret key is encrypted and sent as the message payload, blob name and size are added to the message extra field. If the blob share request is accepted the next message will be the encrypted blob bytes. On the receiving end the bytes are then written to disk and the blob is added to the nodes' list of blobs. The key is decrypted using the nodes own private key, we set it as the blobs key and it can now be decrypted.

Table 3.6: Operations for post blob.

$n_1 - > n_2 : \{m, \{s\}_{p_2}\}^{n_1}$	message with secret key s encrypted by node 2 public key
$n_2 - > n_1 : \{r\}^{n_2}$	response message signed by node 2
$n_1 - > n_2 : \{b\}^s$	blob bytes encrypted by secret key

3.7 TEST APPLICATIONS

With these core modules we can now design example applications that builds on our implementation to achieve secure data sharing between nodes. When implementing test applications using our modules, it is important to keep in mind blob functionality with what we store on disk and where we store it. As you usually will do multiple executions of the same tests which with each execution will store data either in the form of some bytes or certificates, we should clean up any on disk blobs after each execution. While certificates will be overwritten, they should still be removed as if any operation fails during generation or sharing of certificates, it still exists which can (but should not) lead to any unwanted behavior where the nodes old public key from previous execution will not match their new private key.

3.7.1 SHARING DATA BETWEEN NODES

If we have 3 nodes $\{n_1, n_2, n_3\}$, and n_1 wants to share some data $\{d\}$ with the other nodes. n_1 must then make a post call for each node it will share data with. Sharing data requires the recipient nodes to have their servers running to accept messages. The sender must have the certificates of each recipient nodes

(to encrypt the secret key used), and the recipients must have the senders certificate(to verify message signature).

Listing 3.12: Share data multiple nodes

```
node = Node(host, port)
node.get_csr(CAaddr)
for n in nodes:
        node.get_certificate(n)
        node.post_certificate(n)
        node.post_blob(n)
```

3.7.2 SECURE MULTI-PARTY COMPUTATION

Using these modules we can implement secure multi party computation protocols. The nodes share keys through certificates and encrypts data shared between nodes. Data is sent as bytes and the user must control converting data type between nodes. We begin with an example where we have multiple worker (participant) nodes and a trusted third party CA node to issue certificates and compute the function, this example follows the ideal SMC model mentioned earlier. The worker nodes communicate their input to the trusted CA node to compute a function of the combined inputs. This is a simple example to showcase how our PKI and data management design can support a setup for securely sharing data in a network of nodes. Simply put each node provides input, and must receive final output.

Figure(3.2) visualize the example with 5 participant nodes, the lines and arrows represent the direction of which node establish communication (we do expect messages in return). We can see that no participant nodes communicate with each other, and the trusted node is the only node to receive data. This preserves the privacy of input for participants.

First we launch a node that acts as certificate authority. The CA node generates a self signed certificate, and starts a server to listen for messages. We then launch the worker nodes that will compute something on combined input. Each worker node must do the following operations as shown in Listing (3.13) and the trusted CA node shown in Listing(3.14). First we generate a certificate sign request which we send to the CA node to get a certificate signed by the CA. In this example we could create self-signed certificates instead of requiring the CA to generate them, as each node only communicates with the CA anyway. But to show how it would be used in a real SMC implementation that makes use of a PKI we still do it here.

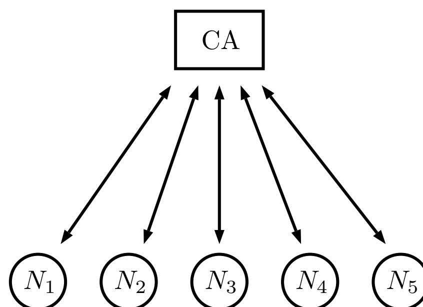


Figure 3.2: CA and participant nodes in ideal model

Listing 3.13: Setup certificates

```
CAcrt = node.get_certificate(CAaddr)
node.get_csr(CAaddr, subject)
node.server_start()
node.post_blob(CAaddr, CAcrt, input)
while len(node.blobs) < 1:
        #wait for result
result = node.blobs.pop()
```

The trusted CA node must then wait until it receives data from participants before it can continue, in our example we loop until the length of the list of blobs is non zero. We can then pop a it from the list. Depending on the function to compute we either wait for all inputs or do part of the computation whenever input is received. In this example each participants input is a number, and the output is the sum of all inputs. Therefor we do not need to wait until all inputs are received. We convert received data to correct data type and add it to the total sum, this is repeated until we reach the expected amount of inputs.
After the sum is computed we share the result with each participant node.

Listing 3.14: Trusted third party

```
for i in range(0, n)
        while len(ca.blobs) < 1:
                #wait for input
        blob = ca.blobs.pop()
        data = from_bytes(blob.load())
        result = result + do_something(data)

for node in participants:
        ca.post_blob(node, nodecrt, result)
```

This example shows that our implementation enables securely sharing data between nodes. Expanding on this we can design systems for managing nodes and distribution of work to preserve the privacy of any nodes input in a real model SMC protocol with no trusted third party node.

Chapter 4

Evaluation

With our design implemented and test applications to show it works as intended, we can now evaluate the performance and correctness. First we look at the system as a whole and then separate the components for individual testing. In capstone project[4] we did a benchmark of NOOP crypto operations and we will refer to those results when discussing performance. All tests are ran on a single machine using python 3.7. The machine runs Windows 10 educational 64-bit operating system, with i5-9400 2.9GHz CPU and 16gb RAM. Each node is launched as a thread with a unique port and name to avoid collision of node identifiers and server sockets. Any benchmark or timed execution is done using NOOPs *Timer* module.

4.1 PERFORMANCE

Our design revolves around nodes and node operations, and we want to evaluate the performance of these nodes. To benchmark the nodes performance we separate tests into the receiving node and sending node. On the receiving end we benchmark the node servers request handling, on the sending side we benchmark the node operations. There are some factors we can disregard, or are limited by based on design choices for implementing certain features (such as standard python modules). We communicate through python sockets and we are not concerned with the transfer speed of the network and socket. We also work with file io for blob management, which although appears non existant to the user, is still a factor in performance of the system. What we mainly want to evaluate with performance is the overhead of our design with additional execution time and memory usage.

4.1.1 TCPSERVER

The node server is built with NOOPs tcp sockets, and each connection is handled in a separate thread. Our performance is therefor limited by pythons socket

and thread implementation. Each thread handling a request may also open files for reading/writing through the use of blobs, file operations are limited by operating systems. Blobs are implemented to not keep file handles open for more than each method call (load, store). In the context of our usage with PKI and SMC we should not hit these limitations until we use a high number of nodes simultaneously connecting to a single node (e.g CA), or when running a high number of nodes as threads on a single machine. New blobs on each node are stored in a separate directory for that node only, so blobs should not perform file operations on the same file simultaneously. The only case would be if two or more nodes on the same machine initializes a blob from a file in a shared directory, and operates on the blob content simultaneously.

To find a rough reference number where we hit the limitation on number of nodes for our system we run a PKI test where we launch 1 CA node and n worker nodes, this is essentially a test to find the number of concurrent connections supported by our TCPserver implementation. The worker nodes all generate a CSR and requests a certificate from the CA. Each worker node performs the operations get/crt (to receive CAs certificate) and get/csr (to get a certificate generated by the CA). We incrementally increase the number of worker nodes n until failure to find the limitations of our implementation.

Listing 4.1: PKI test

```
def test():
        n = 150
        CAnode = Node(host, port)
        CAnode.server_start()

        for i in range(1, n+1):
                t = thread.Threading(target=worker, args=(host,
                    port+i, CAnode.addr))
                t.start()
        time.sleep(10)
        CAnode.server_stop()

def worker(host, port, CAaddr):
        node = Node(host, port)
        node.get_csr(CAaddr)
        node.get_certificate(CAaddr)
```

With up to $n = 150$ worker nodes the test runs smoothly with no errors or warnings, with $n > 150$ we start getting warnings from the NOOP tcp module, the warning tells us that more than one connection is available to be accepted, we found this warning to be a non issue. With this amount of nodes there is a queue for accepting socket connections and the first valid connection is accepted (next connection in queue will be accepted next server iteration). With $n = 200$ we find that connections are being rejected, this is likely because the default number of sockets allowed to queue is set to 1 in NOOPs tcp socket implementation. This is confirmed by increasing the queue number on the TCPservers accept call from default(1) to 10, and we find that this allows us to successfully run the test with up to $n = 300$.

In our example of the ideal model SMC, we can handle as many number of participants as a single node can take simultaneous requests from, as described above. The limitation of the number of participants we can support in a SMC comes from node communication. To allow for more participants we would have to spread out the timing of requests.

4.1.2 BLOB

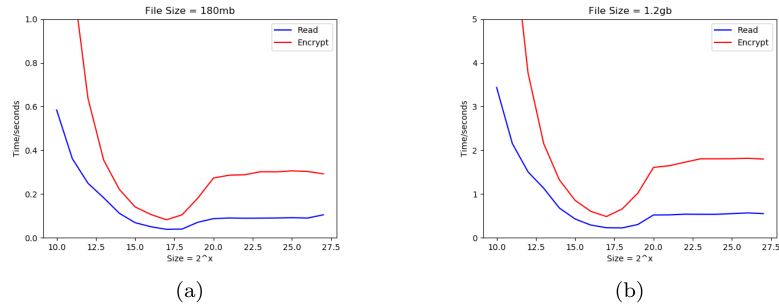
When working with potentially large files for encrypting and decrypting it is not always feasible to hold the entire file in memory at once, especially if multiple nodes run on the same machine. This means that depending on file size we often work on parts of the file at a time. When working with parts of a file, we need to decide on the size of this part. In [4] we found that the performance when working with bytes in python depends on the size of data. Execution time when reading and encrypting bytes varies based on size. This directly affects our blob implementation as we (mainly) store data on disk, and read before encrypting/decrypting this data.

A Graph for these results are shown in Figure (4.1). We measure the time (in seconds) to read and encrypt a file. The X axis shows the chunk size of data we work on at a time(part of total size) and Y axis is time to perform the operation (in seconds). We re-do the test from [4] on python read and NOOP encrypt method with a larger range of chunk sizes, with size ranging from 2^{10} to 2^{28} (1 Kilobyte to entire file) , we ran the test with various file sizes and all showed the same results. The encrypt method in blob will both read and encrypt data. We do not time the execution of the blob methods, but the execution of operations in the blob methods. Code example in listing(4.2) shows the separation of operations and timing of execution using NOOPs timer module.

Listing 4.2: Timed execution of Blob operations

```
f = open("path_to_file", "rb")
key = AESKey()
timer = Timer()
with timer("file_read"):
        data = f.read(SIZE)
with timer("encrypt_data"):
        cipher = key.encrypt(data)
timer.finish()
```

The graph displays peaks in performance with the same chunk size for both file sizes, this tells us that the optimal (at least on our system) chunk size with any file is between $2^{16} = 64$KB and $2^{18} = 256$KB. Interestingly as both read and encryption operations follow the same curve, while one reads from disk and one from memory. From this we can tell that the performance is limited by how python manages bytes, and not by the operating systems file I/O.



(a) (b)

Figure 4.1: Blob operation performance with varying chunk size

Blobs are implemented to not keep file handles open outside methods calls, e.g load/store. This was done for multiple reasons; First is the potential of a large amount of blobs active on a single system. The operating system limits the total number of file handles open at a time, and with potentially multiple nodes with multiple blobs on a single machine at a time this becomes an issue. Second, multiple nodes on a single machine can potentially set a blob with the same path. This is however a user concern as we do not lock a path to an initialized blob, but we aim to avoid any I/O conflict in our blob implementation. This design has the side effect of performance when using load and store part methods. For each call we open a file handle and seek to the position given by method arguments. On method call we seek from start of file, instead of continuing where we last read from (and potentially avoiding seek).

Chapter 5

Discussion

We have designed and implemented tools for a simple PKI enabling secure sharing of data in the context of multi-party computations. We have shown that we can use the tools to implement SMC protocols that preserve privacy for parties involved.

With the node module we provide abstractions for communication and data sharing. Our goal was to provide methods for securely sharing data between nodes with minimal effort of the user. The user of the node need only provide a host, port and subject attributes (which we could base of host name if not provided) to use a PKI or to share data. With this design we can fully focus on the real implementation of an SMC protocol, or any other system using secure sharing of data.

Communication is done with NOOP tcp module for sockets. tcp provides the tcpsend and tcpreceive functions only requiring an address. We found that these were limiting performance as they store opened sockets in a dictionary requiring lookup in each function call. These functions are not designed to be used with a high number of nodes and multiple simultaneous open connections, but for quick and simple communication. In our design we opted for working directly with the socket classes and implementing a server rather than the provided functions to maintain full control over connections for communication and cleanup. This proved crucial when running multiple nodes on a single machine and to scale the design to handle higher number of concurrent connections, where the lists of open connections can take long to search.

5.1 OPTIMIZATION

There are some clear areas in our design that can be optimized. For clarity we implemented certain operations as separate rather than as a single programming abstraction(sharing blob). Receiving and accessing a blob is also implemented in an inconvenient way for the user of our tools.

In our communication implementation with sockets there are some areas we can optimize with increased robustness.

5.1.1 ABSTRACTIONS

When discussing the performance of a protocol in a distributed system, an important factor is number of messages required to complete an operation. In our design we set up secure communication by generating and exchanging certificates. For each node a certificate is generated, and for each recipient certificates are exchanged. These operations are only required one time (unless certificate is revoked or expired). When sharing data the operation requires three messages; sending request, receiving response, sending actual data. For correctness we should also send a response if the blob was successfully received, however this is not implemented.

Generate certificate, exchange certificates, and share blob operations are also not done on the same connection. For each operation we connect the socket, send, and close the socket. This was intentional for the purpose of designing and implementing these operations for a clear separation of the steps required to establish secure communication. It does have clear potential for optimization. In a single method we can check if our own certificate exists (if not, get it generated/issued), check if we have recipient nodes certificate (if not, ask for it), and send our own certificate along with the share blob request message. This abstraction would hide the use of certificates from a user of our tools, and

Listing 5.1: Abstraction of share operation

```
def share(data, nodeAddr)
        if not self.certificate:
                self.certificate = self.gen_csr(CAaddr)
        if not node_crt:
                nodeCrt = self.get_crt(nodeAddr)
        self.post_blob(data)
```

only require us to establish one connection to achieve secure sharing of data. The only requirements to setup secure communication would be initializing the node object with address and port(and optionally CA address), and provide the recipient nodes address. If no CA address exists in a node object we can assume that we generate self signed certificate instead, this could however be a security risk without proper authentication. To complete these operations as one we would need to re implement the node methods to not close the opened socket for both sending and receiving.

When receiving and accessing a blob, the current solution without added abstraction is to loop until length of the node blob list is a certain length. For added programming abstraction we can add this to a node method. Example in

Listing 5.2: Abstraction of receive blobs

```
def receive(num:int) ->list:
        self.server_start()
        blobs = []
        i = 0
        for i in range(0, num)
                while len(self.blobs < 1)
                        #wait
                b = self.blobs.pop()
                blobs.append(b)
        self.server_stop()

        return blobs
```

Listing(5.2) shows how we can implement a method for receiving a fixed number of blobs. The method returns a list of blobs. Wait is usually implemented with a time.sleep to preserve processing usage. The example method starts and stops a server to only receive the specified number of blobs. We can also add a timeout in the case of failure at other nodes we expect to receive from.

5.1.2 ENCRYPTION OPERATIONS

In a previous project[4] we implemented bundling of encrypted secret keys for multiple recipients in the same operation. First encrypting some data ${}_{s}\{d\}$ with secret key s, then encrypting s with each recipient nodes public key $\{s\}_{p_n}$. We can adopt this in our design. We add a list (with node id as key) of these encrypted secret keys to the post blob message, replacing the single secret key payload with the list of secret keys as the payload. Each node would then find the secret key encrypted by their public key by their node id.

Another solution is to instead build a new message for each recipient to maintain the single key payload, this would keep the recipient parsing process the same as if it is the only receiving node of this data. While this method would increase amount of data transferred for a post blob by increasing the size of the message, it would significantly reduce the number of encryption operations required for the sender node as we only use one secret key, this is especially relevant for large files. This approach is useful when sharing results of an SMC where there might be multiple recipient of the same data. Listing(5.3) shows how these encrypted secret keys would be bundled in a single message, the same message would be sent to each recipient. The example is reduced to only showing the message field names, not the contents.

Listing 5.3: Secret key bundling for multiple recipients of same Blob

```
msg = {
        "message": {
                "meta":{},
                "payload": {
                        "node_1":,
                        "node_2":,
                        "node_3":
                        }
                }
        "signature"
        }
```

In our example test application of an ideal SMC protocol, the input of each participant is an integer. The input of a participant can be any data type. We can provide some support for this when sharing data through the use of NOOPs *one* type. The input argument for share blob can then be one of the types set in the function definition. If type of data given is not bytes we can perform some conversion or encoding. The type of the blob data must then be added to the post message to handle conversion on the recipient side back to original data type. We could implement this support for standard data types (int, float, string) or objects with a *__bytes__* method.

5.1.3 SOCKET

If we find the limit in our implementation for concurrent socket connections to be insufficient, we can redesign our communication methods to try again should a connection be rejected by the remote node. The throughput of a socket server would remain the same but this would allow us to manage a higher number of clients without failure.

5.2 PKI

In our simple PKI trust is built between nodes from a shared trusted node. Our PKI follows a centralized hierarchical model of trust. Public keys are shared through public-key certificates signed and issued by the trusted node. The trusted node is in a higher level in the hierarchy or trust. In our implementation there is only one level of trusted nodes, the certificate authority as a root node in the PKI, the root node generates a self signed certificate with itself as both subject and issuer. Secret keys are shared by encrypting the secret key with recipient nodes public key. A new secret key is generated for every blob shared. Certificates are stored on disk in the PEM format when not in use. Asymmetric key pairs are generated (or loaded from disk) when initializing a new node.

Another model of trust is the decentralized web of trust used in the pretty good privacy (PGP)[22] software developed by Phil Zimmerman. As there is no central trusted node each node have to choose who they trust. Each node builds a collection of signatures from trusted nodes who vouch for the validity of

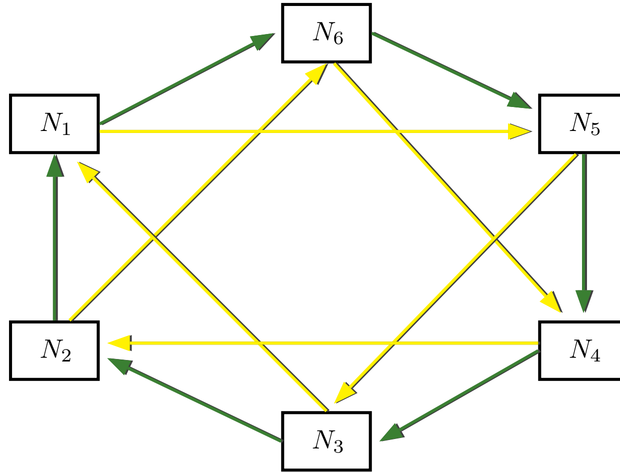


Figure 5.1: Web of trust

a node, building indirect trust in a web of nodes, this grows into a web of trust. Figure(5.1) shows a web with direct and indirect trust displayed as directed lines. Green lines means trust and yellow lines means indirect trust. Compared to the hierarchical model from Figure(2.2) where one node (CA) signs a nodes certificate, the web of trust has potentially multiple signatures to vouch for the trust of a node.

In the context of building trust for participating nodes in a single execution of a SMC protocol, it makes sense to use a shared trusted node and a centralized model. When all nodes in a network are mutually untrusted, there is no initial trust to begin building a web of trust.

5.3 SECURITY & PRIVACY

We provide secure communication through the use of public-key cryptography. Public keys are stored and shared through certificates. Certificates are trusted by verifying the signature of the certificate, signed with the issuers private key. The issuer is a trusted entity in a PKI. We do not perform extensive authentication on the entity requesting a certificate from the CA, apart from verifying the signature of the request message. While any data shared is encrypted between nodes, any node can receive a signed certificate and any node with a signed certificate (by shared CA) can share blob with other nodes. We also sign the full message when communicating, this provides security from injected messages by malicious adversaries. Secret key and blob data is encrypted before sent to another node, providing security from eavesdroppers. Message fields and certificates are not encrypted for transmission. Message fields should not contain

any sensitive information. In our implementation we include original file name and file size in the message, changing the file name is a non issue as the primary purpose is to preserve the correct file extension after transfer.

An important factor that relates to our implemented tools is the existence of semi-honest and malicious adversaries. If our implementation can be used in a way that negatively impacts security, we can not securely support the execution of a SMC algorithm. Our design is simple and naive in that any node following protocol and message format is trusted and can get a certificate issued and signed by a CA. This was a design choice as we expect any full implementation of a protocol using the PKI and tools provided may also do some authentication on the nodes requesting an issued certificate. The node is designed as a base to expand further functionality such as authentication and additional message handling.

Privacy in the context of SMCs is preserved by the protocol used. It is therefor not guaranteed by our tools, but by how our tools are used. Given a privacy preserving protocol, our tools should preserve the security of communication between nodes. Any sensitive data is encrypted for transfer and the protocol decides what data is transferred. Papers with protocols referenced shows proofs on how their protocol preserves privacy with the presence of adversaries and will not be discussed here.

5.4 SUPPORT FOR SMC

As we have implemented tools to support the execution of SMC protocols, we want to discuss how we can achieve this with a real example. In the introduction we stated that our work was in the context of analysis on electronic health records. We can separate this context into two domains of SMCs[13], data mining[20] and statistical analysis[12], both of which are relevant in the context of electronic health records. Secure multi party computations are about preserving the privacy of involved parties.

With data mining we mean each party has their own database where we wish to run some algorithm on the union of those databases, without allowing any participant to view another participants database. With statistical analysis we mean running some analysis algorithm on private data sets of each participant. An example being correlation between age and salary where one party has a data set of age and another has data set of salary. The difference of these two domains are that with data mining we find some data from a query on the combined database of all participants, and statistical analysis we learn something based on the data input.

As already mentioned we provide programming abstractions for SMCs to share data between participants, but a protocol must be implemented to decide what is shared and how we handle what is shared.

In SNOOP[2], the execution of SMC protocols are supported by a coordinator node who prepares the computation and generates a computing graph of

the participating nodes. The graph is an overlay network with directed edges as messages sent between nodes participating in the algorithm. SNOOP is an extension of NOOP, which supports contract based deployments of software components. In SNOOP the contract is used to specify data, services and resources available at a node. The contract maps implementation to names in it namespace, such as encryption and communication methods. Combined the coordinator node and contracts provide a middleware for secure multi-party computations.

In our design we aim to support the development of SMC protocols by extending the functionality of the base Node class. We can extend it to add further abstractions for sharing and receiving data, supporting specific SMC algorithms. We can similarly to SNOOP add methods for deployment of nodes at remote systems. The blob module supports the management of data of any size, this is relevant when working with SMC protocols in the context of data mining databases.

5.5 SMC PROTOCOLS

We want to mention some protocols in the domain of statistical analysis and data mining, with what they want to learn from the computation. We will not discuss proof for security or privacy from the computation, that is found in referenced articles.

Protocols for statistical analysis[12]. For statistical analysis the goal of the computation might be to learn of the mean, variance, or regression line of the combined data set of inputs from each participant. This is usually achieved with some circuit evaluation protocol, an example being 1-out-of-N oblivious transfer by Goldreich[16].

Protocols for data mining[20][21]. There are two classic settings for privacy-preserving data mining. First, data mining algorithms on the union of participants databases without letting any party view another participants private database. Second, analysis on some statistical data from the database which should still preserve privacy while being able to obtain meaningful results from data mining algorithms.

Protocols based on homomorphic encryption[23][18][19]. Homomorphic encrypryption is an interesting field with regards to SMCs. With homomorphic encryption we can perform operations on encrypted data without decrypting it first, preserving the privacy of input. In a SMC protocol with homomorphic encryption, each participant encrypts their input, then perform the computation on the ciphertext inputs, and finally a distributed decryption on the final ciphertexts to get the results.

Chapter 6

Conclusion

Based on requirements for secure multi-party computations we have designed and implemented a set of tools based on the NOOP and SNOOP middleware to enable secure communication of participants and preserve the integrity and privacy of data exchanged. This is achieved through the use of secret key and public-key encryption, and a simple PKI to generate and issue signed certificates containing the credentials and public key of a node. Certificates signed and issued by a trusted certificate authority enables trust between participant nodes of an SMC by verifying the certificate authorities signature. Combined with secret key encryption and public keys shared through certificates we can securely share data between nodes.

We evaluate and discuss some limitations to our design and implementation and potential optimizations for future work. Higher level programming abstractions are suggested to hide the use of cryptography and certificates when communicating with other nodes and sharing data. Our work is in the context of electronic health record data, and we discuss the type of secure multi-party computation protocols relevant to this domain and how our implementation can support an example protocol.

Bibliography

[1] Andersen, A., 2013. Types, signatures, interfaces, and components in NOOP: The core of an adaptive run-time. In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA) (p. 1). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[2] Anders Andersen. Types, signatures, interfaces, and components in NOOP:The core of an adaptive run-time. InThe International Conference onEngineering of Reconfigurable Systems and Algorithms (ERSA 2103), July2013.

[3] Andersen, A., Yigzaw, K.Y. and Karlsen, R., 2014, October. Privacy preserving health data processing. In 2014 IEEE 16th International Conference on e-Health Networking, Applications and Services (Healthcom) (pp. 225-230). IEEE.

[4] Cloud security infrastructure and NOOP, Capstone 2018

[5] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R. and Polk, W., 2008. RFC 5280: Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile. IETF, May.

[6] Ellison, C. and Schneier, B., 2000. Ten risks of PKI: What you're not being told about public key infrastructure. Comput Secur J, 16(1), pp.1-7.

[7] Cryptography, Python Package, https://cryptography.io/ 12.15.2019

[8] Kent, S.T., 1993. Internet privacy enhanced mail. Communications of the ACM, 36(8), pp.48-60.

[9] Linn, J., Barker, C., Bidzos, J., Bishop, M., Cohen, D., Fox, C., Gasser, M., Housley, R., Kent, S., Laws, J. and Lipner, S., 1993. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421.

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

[11] Maurer, U., 1996, September. Modelling a public-key infrastructure. In European Symposium on Research in Computer Security (pp. 325-350). Springer, Berlin, Heidelberg.

[12] Du, W. and Atallah, M.J., 2001, December. Privacy-preserving cooperative statistical analysis. In Seventeenth Annual Computer Security Applications Conference (pp. 102-110). IEEE.

[13] Du, W. and Atallah, M.J., 2001, September. Secure multi-party computation problems and their applications: a review and open problems. In Proceedings of the 2001 workshop on New security paradigms (pp. 13-22). ACM.

[14] Du, W. and Zhan, Z., 2002, September. A practical approach to solve secure multi-party computation problems. In Proceedings of the 2002 workshop on New security paradigms (pp. 127-135). ACM.

[15] Yao, A.C., 1982, November. Protocols for secure computations. In 23rd annual symposium on foundations of computer science (sfcs 1982) (pp. 160-164). IEEE.

[16] Goldreich, O., 1998. Secure multi-party computation. Manuscript. Preliminary version, 78.

[17] Cramer, R., Damgård, I. and Maurer, U., 2000, May. General secure multi-party computation from any linear secret-sharing scheme. In International Conference on the Theory and Applications of Cryptographic Techniques (pp. 316-334). Springer, Berlin, Heidelberg.

[18] Cramer, R., Damgård, I. and Nielsen, J.B., 2001, May. Multiparty computation from threshold homomorphic encryption. In International conference on the theory and applications of cryptographic techniques (pp. 280-300). Springer, Berlin, Heidelberg.

[19] Damgård, I., Pastro, V., Smart, N. and Zakarias, S., 2012, August. Multiparty computation from somewhat homomorphic encryption. In Annual Cryptology Conference (pp. 643-662). Springer, Berlin, Heidelberg.

[20] Lindell, Y., 2005. Secure multiparty computation for privacy preserving data mining. In Encyclopedia of Data Warehousing and Mining (pp. 1005-1009). IGI Global.

[21] Lindell, Y. and Pinkas, B., 2000, August. Privacy preserving data mining. In Annual International Cryptology Conference (pp. 36-54). Springer, Berlin, Heidelberg.

[22] Garfinkel, S., 1995. PGP: pretty good privacy. " O'Reilly Media, Inc.".

[23] Gentry, C., 2009, May. Fully homomorphic encryption using ideal lattices. In Stoc (Vol. 9, No. 2009, pp. 169-178).