



UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Sneak

A secure multiparty computation module for Python

Torkel Syversen

INF-3990 Master's Thesis in Computer Science - [May] 2023

Abstract

Secure multiparty computation (SMC) is a technique that allows multiple parties to jointly compute a function while keeping their inputs private. This technique has gained significant attention due to its potential applications in various fields, including privacy-preserving healthcare, politics and finance.

SMC involves a set of protocols that enable parties to achieve secure computation and analysis. These protocols typically involve a trusted third party or a cryptographic algorithm that ensures the privacy of the inputs. Some popular cryptographic algorithms used in SMC include homomorphic encryption, secret sharing, and the one discussed in this thesis, denoted as the round-robin scramble.

This thesis focuses on the realization of a secure system for analysing sensitive data across multiple nodes in a distributed network. The thesis discusses the approach, design, and implementation of such a system with emphasis on security, usability, and scalability. Security is of upper importance to prevent information disclosure, followed by usability to ensure practicality and ease of use. Scalability is addressed to accommodate networks of varying sizes. The proposed system, named Sneak, offers near-zero information disclosure by leveraging Python, enabling robust and valid complex analysis operations across distributed networks.

Contents

Abstract	i
List of Figures	v
List of Tables	vii
1 Introduction	1
2 Related work	5
2.1 The millionaires problem	5
2.2 A practical approach to solve SMC problems	6
2.3 Secret sharing	7
2.4 Snoop middleware	8
3 Concepts	9
3.1 Secure multiparty computation	9
3.2 Coordinator	12
3.3 CIPHERING data	12
3.3.1 Encryption	12
3.3.2 Symmetric	13
3.3.3 Asymmetric	13
3.4 Digital signature	14
3.5 X.509 certificate and Certificate Hierarchies	14
3.6 Server	15
4 Architecture and design	17
4.1 Initialising the node graph	18
4.2 Security requirements	19
4.3 Minimum nodes needed	27
4.4 Sneak communication	27
4.5 Cryptography module	28
4.6 Setting up servers	29
4.7 Shutting down server	31
4.8 Running SMC operations	32

5	Examples and experiments	37
5.1	The fair competition problem	37
5.2	Performance	44
6	Discussion	53
6.1	Experiments	53
6.2	Library security	54
6.3	Library simplicity	55
6.4	Library scalability	56
6.5	Computational issues with Sneak	57
6.6	Preventing man in the middle attack	58
6.7	Onion encryption vs partial encryption	59
6.8	Future work	59
6.8.1	Automacy	59
6.8.2	Databases	60
6.8.3	Executable code	60
7	Conclusion	61
	Bibliography	63
A	Appendix	67

List of Figures

3.1	SMC algorithm	10
3.2	Encryption to ciphertext	13
3.3	Digital signature verification	14
4.1	Node graph illustration	19
4.2	No encryption	20
4.3	N_a injection	20
4.4	Eavesdropper listening	20
4.5	n_a injection, encryption but no signature used	21
4.6	SMC: Signing data	22
4.7	SMC: With scramble	24
4.8	SMC: Scramble issue	24
4.9	Unique scrambles	25
4.10	Unique scrambles issue	26
5.1	Alice's server	38
5.2	Bob's server	38
5.3	Charlie's server	38
5.4	Coordinator's server	39
5.5	Alice's server	40
5.6	Bob's server	41
5.7	Charlie's server	41
5.8	Coordinator's server	42
5.9	Contestant received their vote result	42
5.10	Sequential requests running on a simulated Sneak SMC network on a local machine. First image is zoomed in.	45
5.11	Concurrent requests running on a simulated Sneak SMC network on a local machine.	46
5.12	Concurrent requests running on a Sneak SMC network on the UiT cluster.	46
5.13	Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.	47
5.14	Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.	47

5.15 Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.	48
5.16 Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.	48
5.17 Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.	49
5.18 Time used on coordinator to initialize a request compared to the total time used on a request. (On local machine)	50
5.19 Time used on coordinator to initialize X concurrent requests compared to the total time used on X concurrent request (on local machine).	51
5.20 Time used for a single request with/without compression. (On local machine)	52

List of Tables

4.1 List of denotations	17
-----------------------------------	----



Introduction

Data can be anything, including sensitive personal information. For hospitals, this type of data is health records and is essential in order to diagnose and cure diseases. This is just one type of sensitive data in a sea of many. The common factor is that sensitive data should not be shared with anyone without specialized granted access, both for safety and personal purposes of patients. However, data is the foundation for which further knowledge can be extrapolated. Combining the data from different sources is crucial for an increased individual or group knowledge in a distributed setting. As Covid-19 showed, being able to track the spread of the disease is very important in the event of a pandemic. To share the sensitive information about where you have been in a secure way, can therefore help reduce the spread of a highly contagious disease. The privacy concerns here can be questionable. Therefore, to analyse patients health records introduces privacy concerns that undoubtedly must be respected. If health records for patients are scattered between many institutions, many cities and potentially many countries. Their personal records will be data sets located on different databases and machines. Since jointly analysing this data imposes ethical and legal privacy concerns, it poses the question; How can a patient's sensitive health records be analysed if the data sets are scattered on different institutions, while still not disclosing the data sets to any other institutions? There should be at all times only one institution that can read and analyse the health records of patients and that is the institution who stores it. There is none to the exception. By relaxing this constraint will potentially allow for leak of information, which can have social impacts on whom it concerns. Preventing leak of information is therefore the overall goal when analysing

distributed data sets of this kind.

Secure multiparty computation(SMC) is a way of achieving this goal. With nodes (computers) connected together in some form, creating a network of communicating computers. Each node will perform their own analysis on their own data. Add their data to a result sum and pass it on to the next connected node in the network. Eventually, all of the nodes will have added their analysis result to the overall result. The fundamental idea is assuring that no one else will have access to ones sensitive data. Having this strict rule makes it complicated to share the result of the computed analysis data. Often the result itself is sensitive too and must be treated as such. For instance, if it's possible to deduce sensitive information from the result of one single node, then the algorithm discloses information for that node. However, add enough results together and it will be safe to share the unified sum of results as it will be impossible to deduce any individual node's result from the sum. The result sum mentioned here effectively means the result of the analysis performed across all nodes. Which in theory could be anything, a single percentage value, to a complex matrix of values.

To give an example I introduce the analogy of *the fair competition problem*, which is similar to the millionaires problem [1]. It contains the SMC specific problems with relation to joint analysis of distributed data sets.

A singer competes in a contest where the winner gets fame, money and glory if they manage to win. The competition consist of two judges who must both agree yes for the singer to continue to the next stage. The constraint is that the votes must be anonymous, judges cannot know the other judge's decision in order to keep the contest as fair as possible for the people who compete. How can the judges reach a decision on whether or not the singer moves on to the next stage of the competition?

Using the SMC algorithm from this example, if one judge votes yes and the other votes no, the singer will not pass through. The judge who voted yes can then deduce that the other judge must have voted no, as there is no other possible outcome for the result that occurred. Data has therefore been disclosed from one of the judges. The judge who voted no, will however not know what the other judge voted for. Nonetheless, if we change the number of judges from two to three, the votes can no longer be deduced from the results, thus there is no more obvious information disclosure from this example. It is however questionable if three judges is enough to effectively hide the individual vote results, but these are special cases. This is the premise that allows this SMC algorithm to work as data is not disclosed between the connected nodes, as long as there exist enough nodes in the network. Data can therefore be analysed and shared securely. Notably, in the *fair competition problem*, a judge's vote is regarded as sensitive (and hidden) information.

In this thesis, how to realise a system to perform analysis on sensitive data involving multiple nodes across a distributed network, in a secure manner, is investigated. The thesis is based on some early work from the SNOOP [2] platform. Moreover, in this work, explaining how to approach, design and implement such a system is discussed in more detail. The important factors this implementation focuses on will be (i) security, (ii) usability/simplicity and (iii) scalability.

- i There is no surprise that security is the most important factor as the algorithm would be meaningless if not properly secure. SMC must be ensured to not disclose vital information.
- ii Usability is the number two concern as to make it practical and easy to use. A complex system is often tedious to work with and can be quite error prone if not handled correctly. A user who doesn't understand the usage could accidentally make it less secure or accidentally introduce loopholes to its sensitive data. A Python library that is simple is therefore preferred.
- iii SMC networks might differ in sizes. Three connected parties will by default work faster than 20 connected parties. The scalability issues should not be incomprehensible in such cases. Depending on the involved parties, the algorithm should work within a feasible time frame, to make it a realistic and practical library module.

Ideally we want zero information disclosure for a SMC algorithm. However, this can prove quite difficult to achieve in a practical sense. This is due to nodes that can have malicious intent and communicate with each other outside the scope of the library module. Therefore, near zero disclosure has been designed and implemented leveraging Python in a way to hide complexity and maintaining the security requirements. It will be a library module capable of performing SMC across a distributed network of computers, each containing sensitive data. Furthermore, it will support a wide range of complex analysis operations, whether it's uniquely defined for separate nodes or consistent. Any complex result data can therefore be calculated, making the system robust and valid for most use cases. We introduce Sneak, a practical approach for realising a SMC algorithm.



Related work

2.1 The millionaires problem

The problem of secure multiparty computation was first introduced by Andrew C. Yao in 1982 [1]. In his paper he introduced the millionaires problem, which is a simple yet insightful example of a specific SMC use case. However, because of the prerequisites, the problem proves quite difficult to resolve. The millionaires problem involves two rich millionaires who wish to know who is richer between the two. The prerequisite is that they can't tell each other how much money they have to the other person. Additionally, even though it's not mentioned in Yao's paper, they can't tell anyone else about their wealth either. In such a case, how can they carry out their conversation and reach a conclusion? In a practical sense it seems impossible, but it can be achieved leveraging mathematical expressions.

What Yao proposes is essentially, hiding the data by using one-way functions, which is a way to make it easy to calculate a result but hard to know what input was given to produce the result (invert the function), thus promptly named one-way functions. To give an intuitive and over simplistic example of how it works with the best of my understandings. The two millionaires asks each other; do you have X million? Do you have $X+1$ million? Do you have $X+2$ million? And so on, until one of the millionaires says yes. Then the other can know he has more or less millions of the two. This assuming their wealth is differentiated in the millions. X here is any positive number. For the sake of a simple example, it's not completely accurate, but is the foundation of how it

works. By using the one-way functions, they won't know what specific number they ask for, but they know if they are above or below it which is how they are able to deduce if they are richer or poorer than the other. It also makes it very difficult (but not impossible) to disclose information to the other person.

Yao's paper is mentioned as it is a fairly different implementation from what is proposed in this paper but nonetheless as valid. It simply applies to another branch of SMC problems. There are many different SMC specific problems and not all of them can be solved by a general SMC implementation, at least not efficiently [3].

2.2 A practical approach to solve SMC problems

Wenliang Du and Zhijun Zhan discusses various specific SMC problems and mentions that it's hard to create a practical implementation which handles all of them in a general form. They discuss what is referred to as the secure two-party model. *If the two-party model can provide a practical solution, we do not need another model. However, according to our past experience, efficient solutions for this model are usually difficult to find* [3]. The millionaires problem mentioned above is one example of a two-party problem. If a good practical and general solution to this could be formed, then it should be able to solve most if not all other SMC problems as well. However, since this is not the case, practical solutions to SMC problems is therefore usually developed for specific problems.

Their paper *a practical approach to solve SMC problems* [3] was released in 2002. They write that achieving an ideal security model is not difficult but achieving it efficiently is. Additionally, using a general solution for special cases of multi-party computation can be impractical. Special solutions should be developed for special cases for efficiency reasons [3]. Their paper allows for efficiency gains by sacrificing some security in an acceptable way. Parameters can be tuned which can allow for higher or lower security measures. It is achieved by using data disguising techniques, such as *polynomial function disguise* and *linear transformation disguise* with the combination of a commodity server. The commodity server is not part of any computations, therefore not responsible for any disclosed information, but does provide data in order to hide data for participants. They also note that practical solutions to the ideal model might not exist. The ideal model is where no data is disclosed to any connected or third party programs, while still being optimal enough to run over a distributed network of nodes in real time.

Such systems must use secure communication, and safe encryption schemes on

top of an SMC algorithm. Depending on the size of the data being transmitted, this can prove quite inefficient and difficult. Optimization techniques such as compressing data can be used but might not be enough. Importantly, compressing data should always be done first as then it will be less to encrypt/decrypt. In these cases, special solutions can be made to sacrifice some security if it's still within an acceptable rate. All depending on what the end user need in these cases. This means that their solution can partially disclose information, but does so to improve performance. The amount of disclosed data can be tuned for special cases to still keep it secure enough.

A similar concept to scrambling data is mentioned in [3]. This concept is adding bogus data into a database. It's another disguise technique where if one can't distinguish between the bogus records and the real records, then the data would be hidden. Whether or not this is effective and secure might be questionable and quite situation dependent but it's definitely an interesting option worth mentioning. It also somewhat similar to Sneak's round robin [4] scramble, which will be detailed in the design section.

2.3 Secret sharing

Shamir's Secret Sharing (SSS) is a useful technique within SMC as it has mathematical properties which can guarantee that a secret which has been split into n keys, can be reconstructed based on a minimum threshold k , denoted as a (k, n) threshold scheme from [5]. If any one key is leaked, it does not disclose any information about the secret. For example, if $k=2$, usually denoted as a $(2, n)$ scheme, at least 2 out of n shares must be present in order to gain the full secret. Secret sharing is especially useful where a single point of failure becomes an issue, such as if a node becomes unavailable, the secret can still be reconstructed by the other k nodes.

It works by generating k random points on a 2D plane, with $k=3$, the line created by these 3 points will define a polynomial of degree $k-1$. Any n number of keys can be constructed on the polynomial line, and any 3 out of n points will together define the polynomial, which will regain the secret for the $(3, n)$ scheme [5, 6].

Many secret sharing techniques are based on SSS. The effects this scheme has on SMC is allowing joint computation of data. For example, this can be used by SMC algorithms to jointly compute the average, if Alice has secret value 102, she can make two shares, (54, 48) which sums up to 102. Bob has secret value 765 and two shares (249, 516). If Bob shares 249 with Alice and Alice shares 54 with Bob, neither party knows the other participants secret value. However,

they can compute the average of both their secret value by combining their shares. Bob will have $54 + 516 = 570$ and Alice will have $48 + 249 = 297$. The average of this data becomes $(570 + 297)/2 = 433.5$, which is equal to the average of their original secret value, $(765 + 102)/2 = 433.5$. However, this comes with computational costs which is why newer techniques, such as [7, 8] aims to optimize it.

2.4 Snoop middleware

One practical approach to solve many SMC problems related to health services is the SNOOP middleware [9] by A.Andersen. Another SNOOP paper [2] mention that databases often are partitioned vertically and horizontally, such that data is contained on many distributed nodes. Performing calculations on these sets can therefore raise a number of security concerns if the data is marked sensitive. Even though data might relate to the same person, these concerns must be held appropriately. In health community services, patients can have many general practitioners who each store data about their patient. SNOOP mentions that for these cases, *"the legal, ethical and privacy aspects at managing those data sets have to be respected"* [2]. In fact, this paper and implementation is based on SNOOP, showing a practical approach to how such a system can be implemented.

The basis for the SNOOP middleware is this. *"The combination of SMC algorithms and Public-key encryption (in combination with symmetric key encryption) ensure that each node is unable to learn about the other nodes local data, input data and intermediate results. A PKI and its certificate authorities (CAs) are used to ensure that the participants can distribute and trust public keys. The PKI enables public-keys as the tool to authenticate participants and maintain the integrity and privacy of the data exchanged."* [2] By using public keys, certificates and CA's, the SMC algorithm is secure to be used in a real world setting. It can be used to perform joint computation on distributed data sets, preserving the legal and ethical privacy concerns.

/3

Concepts

3.1 Secure multiparty computation

SMC is generally described as a cryptographic technique designed to provide privacy and security in situations where parties do not fully trust each other or where the data being used is sensitive or confidential [10, 11]. With SMC, parties can perform computations over their data without revealing it to the other parties or to any external entities, using various cryptographic techniques such as encryption, secure communication channels, and the use of trusted third parties like Certificate Authorities (CAs) to establish trust.

In SMC, no data should be leaked or disclosed to another party under any circumstances. In other words, SMC algorithm is a guarantee that data remains private for the respective participating nodes. The main idea is to be able to perform some analysis individually on a per node basis, and pool the individual results together to form a sum result. The process of adding the performed analytic data to a sum, is what is most difficult to achieve successfully while still maintaining an optimal security certainty. $Node_1$ cannot simply add the result to a sum and further the sum to $Node_2$. Then $Node_2$ will know the sensitive information disclosed in the sum from $Node_1$ which is in violation of the SMC presumption. In worst case scenario, $Node_2$ could deduce the sensitive information from the result sum it received. It is only after a certain amount of results have been added together safe to assume that it's impossible to deduce any information from the result sum. Theoretically, to get to this step is not too difficult, however practically, it's more to it than meets the eye. This will be

explained in more detail under the *architecture - security requirements* section where the specific issues are highlighted.

Essentially, a coordinator node must start off the algorithm and preset the result sum to a random scramble value. As figure 3.1 shows, $Node_2$ will not be able to know $Node_1$ result, but it's still able to add its own analysis result to it. This will remain the case for all nodes the result passes through. Once passed through every node, it's sent back to the coordinator who can subtract the random scramble value. The coordinator is left with the true result from all nodes participating, and no nodes have received any others disclosed data. Figure 3.1 shows an example of the use of scramble value and furthering data from one node to another.

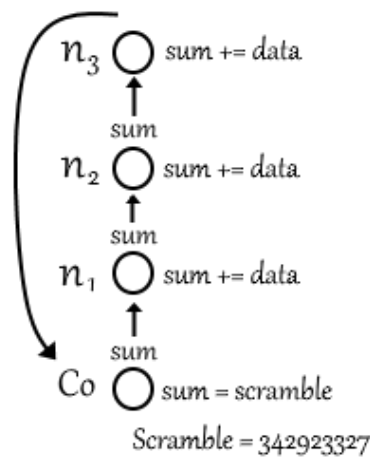


Figure 3.1: SMC algorithm

At this point the sum can be freely and safely distributed, as previously stated this can only happen when there has been enough results added together. One example of this is the *fair competition problem*, where individual votes should not be known, but the total voting result has to be in order to know who won. Meaning that the result is publicly available but a judge's individual vote are not. Another example is the election infrastructure (for instance the presidential election process) which works similar but on a much larger scale.

In some cases within SMC, the result does not have to be shared and can still be kept private. An example of this is when analysing relations between data sets. If node n_a wants to know the relation it has with n_b and n_c then only n_a needs to know the SMC result. n_a can in this case be the coordinator as well as a regular node in the SMC network. This will ensure that the result is only available to one node.

Why do we need secure multiparty computation algorithms? Secure multiparty computation has many use cases in the real world. Especially when keeping information private is as crucial as it is today. In an election system, only the overall result should be public and the individual votes should remain a secret. In the case where someone is able to deduce one's individual vote, suddenly there is a possibility to sell votes which can skew the election in favor of one party or politician.

In auctions, knowing who bid what for which item can raise or lower its value. Therefore using SMC can be a good option to prevent information being public which keeps the auction fair for both sellers and buyers [12].

A group of people can sign a document without knowing who or which other person has signed it. Then the document only passes if it exceeds a given threshold of signatures [12]. If passed or not, the anonymity will still remain.

In health information, SMC is a viable option to perform analysis. Often Pearson's R coefficient is used to calculate the correlation between data. For instance, this could be used to determine whether someone has Covid-19 or is more likely susceptible to Covid-19 by looking at their health records. Data sets with coherent data correlations is very viable in a SMC context for sensitive data.

Additionally, it can be used to track the spread of a disease and to prevent it further. For example there can be a SMC app on mobile phones, each phone will then be a node in the SMC network. If there has been an outbreak in a supermarket X , and an authority wants to know how many is potentially infected by being at the store. Then instead of sending GPS data about where the person has been, it can calculate whether or not the person (phone) has been to X within a given time frame, completely anonymously. It would otherwise not be anonymous if the authority asked the person one by one whether it has been at X .

Tracking using location data is under law sensitive information and can not be done without consent [13, 14]. Whenever location data is used, it can be sent to a centralized server for storage and analysis. For instance to your google account [15]. Applications (apps) can ask to use the location data in some way. Google Maps is a default app on android smartphones and is one example of this usage. It can be hard for users to keep track of where all their data are being stored on the internet, as well as being uncomfortable. While SMC can't prevent the provider storing your location data. It can prevent applications leaking information, and prevent unnecessary storing of this data by keeping users anonymous. This is by not sharing the result at all, but rather perform all calculations locally on the phone in the context where it is applicable and

anonymously share the result if needed. Another situation where this could be used is anonymously study behaviour patterns of people (or a certain group of people), by looking at where they go and where they have been.

Applications could need less consent, as SMC will guarantee the safe handling of information. Users will be guaranteed that their data is not spread or stored somewhere on the internet by the use of the algorithm. Location, and health information is very sensitive for many and should not be public knowledge. Having a secure way to perform analysis is therefore the motivation for Sneak and for reliable SMC algorithms in general.

3.2 Coordinator

A coordinator is also known as a master or leader node. It is a node in a SMC network that is responsible for coordinating the computation process between different parties or nodes. The coordinator's role is to ensure that all parties follow the same protocol and that the computation is performed correctly, securely and efficiently. The coordinator is typically the first point of contact in the system, and is responsible for distributing and verifying certificates to ensure that nodes are who they claim to be. The coordinator may also perform other tasks, such as generating or distributing keys, managing inputs and outputs, and verifying computation results. The coordinator is a critical component of a SMC system that helps to ensure the security and integrity of the computation process. Coordinator can act as any other SMC node but is agreed by everyone in the network to be the leader.

3.3 CIPHERING DATA

3.3.1 Encryption

Both symmetric and asymmetric encryption is used. Symmetric is used to encrypt large data in an efficient way. The problem is that the symmetric key is generated for each message and the other party doesn't know it. This is why the key itself is sent with the encrypted data. The key must therefore also be encrypted asymmetrically using the receiver's public key within the RSA encryption structure. Asymmetric encryption should not be done on large text as it is generally considered slow. However, for a small key of fixed size it seems perfect. This ensures that the data is only readable to the intended receiver.

$$\boxed{\text{Data}} + \boxed{\text{Symmetric key}} = \boxed{\text{Ciphertext}}$$

Figure 3.2: Encryption to ciphertext

3.3.2 Symmetric

When dealing with symmetric encryption AES is a good algorithm. AES has different encryption modes and it's important to not use ECB mode. ECB is the simplest mode, it encrypts identical plaintext blocks to identical ciphertext blocks [16, pp.48]. This means that the algorithm reveals patterns even after it has been encrypted, making it possible for an attacker to understand and possibly retrieve data. Instead, CBC should be used, this mode does not suffer from the same fault by using an initialization vector (IV). The IV is a randomly generated 128 bit number, unique for each cipher. The IV is needed along with the secret symmetric key when encrypting/decrypting data. However, the IV is not meant to be secret as with the key, it only obscures the data. Therefore, when sending encrypted data, the IV must be sent with it in order for the receiver to decrypt it. An Attacker will know the IV, but as long as he doesn't have the secret key, it will not matter.

3.3.3 Asymmetric

RSA is an asymmetric encryption/decryption algorithm. It scales and handles the key-distribution problem natively. Each user generates a public and a secret key. Anyone can encrypt data using the receiver's public key, and only that receiver can decrypt the data. It scales because if some attacker manages to get the secret key to one user, they only have access to that user's secret data [16, pp.53]. Additionally, each participant generates the keys themselves and there is no need for a key distribution middleman.

The problem is nonrepudiation. When sending one message, it's difficult to prove the author of the message if there are more than 2 parties involved. If Bob encrypts a message to Alice, Alice can say she created the message and furthers it to Charlie. It will be troublesome for Charlie to validate who the original author was. Luckily, certificates based on digital signatures and a CA handles this very issue and is therefore also important in the context of an SMC algorithm.

3.4 Digital signature

Digital signature is used to verify the message that is sent. It makes it possible to know who sent the message and whether the message has been modified in transit (by an attacker). The concept of signature works with the sender hashing the entire message content (into lets say a 256 bit value), the hash value is then encrypted using the senders own secret key (which forms the signature) and sent along with the actual encrypted message. Anyone who knows the senders public key can then decrypt the hash value (opposite of how RSA encryption/decryption works). And create their own hash value from the actual received and decrypted message content, and then compare the two. If the hash values match, then they know that only the correct sender has sent this message and that it has not been modified during transit. This is known as they used the senders public key to decrypt the signature, rather than using their own secret key. This means that no one else could possibly encrypt the message without having the senders secret key.

Digital signatures is very important when it comes to SMC as it adds a layer of integrity to each and every message. Preventing any unwanted or invalid messages both due to hackers/exploiters and possible glitches when sending data over the internet.

With digital signatures we know that the sender is the owner of the private key, but not necessarily who he is.

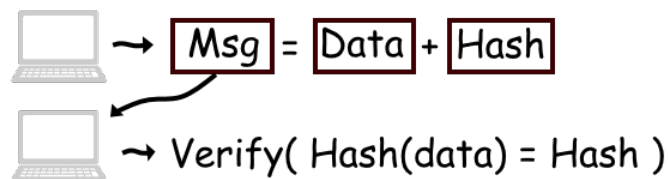


Figure 3.3: Digital signature verification

3.5 X.509 certificate and Certificate Hierarchies

A certificate is the combination of an identity and a public key, signed by a trusted authority. This will ensure that the public key is belonging to that identity.

An X.509 certificate contains information about the identity of a computer or entity, and is widely used to establish that a public key belongs to a given computer [17], as certified by a certificate authority (CA). The CA is a trusted entity that distributes certificates to ensure that computers are who they claim to be. When buying a new computer, a certificate is typically integrated into the system, which is the base of a certificate hierarchy. The certificate is also automatically renewed at regular intervals to ensure that it remains valid [18].

Certificate hierarchy refers to the structure of trust between CAs and end entities (computers). A certificate hierarchy can be thought of as a tree structure, with the root CA at the top and the end entities at the bottom. Intermediate CAs can also be present in the tree, which sign and issue certificates to other CAs. The root CA is at the top of the hierarchy and is responsible for issuing certificates to intermediate CAs. Intermediate CAs can then issue certificates to other intermediate CAs or end entities. This creates a chain of trust that can be verified by following the chain of certificates from the end entity up to the root CA [18].

To have a certificate signed by a trusted authority can potentially involve physical validation. If the CA is Helsenett and if they sign a coordinator's certificate, then each node receiving requests from this coordinator know they can safely trust it. Subsequently, each request from the coordinator must contain their valid and signed certificate, otherwise the request must be ignored by all nodes. Certificates adds a layer of security as it's then possible to assume the coordinator will always be a trusted node. Having this assurance is the only way to make Sneak completely integrity safe, as the coordinator can potentially gain knowledge of other nodes sensitive data otherwise.

Using a self signed certificate will guarantee that the certificate (and other data) has not been modified during transmission. The contents of the certificate will therefore be valid in terms of what the sender sent. However, with self signed certificate anyone can generate it and sign it themselves on their own private key. Theoretically any machine (or node) can pretend to be another, as long as their certificate states it. This is why self-signed certificates are insufficient for an SMC algorithm, and why it is necessary to have certificates signed by a trusted CA.

3.6 Server

The server mentioned in this thesis is the foundation for which communication is performed. It is a way to receive and send messages between participants in the SMC network. For this server, communication is achieved by sending

TCP packets over the internet to the correct recipient. Each TCP packet carries information and will be from now on referred to as requests. For instance, node *A* can send a request (a TCP packet with information) to node *B*. The TCP protocol guarantees that requests will be sent, which is implemented by the OS [19]. Our server therefore does not need to worry about the reliability of sending data over the internet as it is handled for us. One important thing to keep in mind is that, although TCP guarantees that the full message will be sent, it can't guarantee that the request is actually received on the remote end. The remote server might be down or have crashed due to an unknown reason. This specific case is not a big issue for the implementation of a practical SMC library. The result will be invalid and the SMC system will throw an error, letting the end user know about their unattainable server. To make the SMC library as little complex as possible, it is run as a background server, hiding it as well as its functionality from any user.

/4

Architecture and design

Table 4.1: List of denotations

Denote	Description
whitelist / nodefile	Text file of allowed nodes
n	Node / computer
Dirty node	A node which has malicious intent
Clean node	A node which is compliant
Co	Coordinator
m	Message
s	Scramble
m^a	M encrypted to node a
m_b	M signed by node b
m_b^a	M encrypted to a and signed by b

The design details of a Python developed SMC library is discussed here. Detailed examples of usages however will be presented in the experiments section.

Since one of the main goals of this library is to keep it simple to use while preserving the fundamentals of SMC. In short, there will be a background process which runs a server that is used to connect nodes together. Each node in the distributed network has a server to handle HTTP requests. It operates as a background process which means that the user does not have to think, worry or even know that it exists, it just works. This will only be the case after the node has been initially setup and run.

The Sneak client is like the front-end in this Python library and is an intermediate to the underlying SMC network. It communicates with the background server in order to send messages over the network in a secure SMC session. The Sneak client is therefore used when starting and ending SMC operations, as well as retrieving the results.

4.1 Initialising the node graph

The node graph, also referred to as *send_list* is the order of which analysis operation is performed for each subsequent node. It's initialized by Coordinator and its content is partial encrypted to each node. An illustration of the generated node graph is shown in figure 4.1 with 4 nodes and 1 Coordinator node. To make the understanding of a node graph clear, it's a list of nodes containing the corresponding nodes address (IP-address and port). When a node receives the node graph, they pop the first element in the list to gain information (IP-address+port) of the next node in the system and can then further the SMC operation to this next node.

Another example of a possible node graph from figure 4.1 is $[n_b, n_d, n_a, Co]$. *Co* sends data to n_c , n_c sends data to n_b , n_b sends to n_a , and n_a sends the final result back to *Co*. As you can see, node n_c is not in the list. This is because Coordinator knows to send n_c the request and already has its address. In theory the initial node graph could also be written as; $[n_c, n_b, n_d, n_a, Co]$ or even $[Co, n_c, n_b, n_d, n_a, Co]$, but this additional information is redundant.

In figure 4.1, the message m_{Co} contained from coordinator is each individual node-address, partially encrypted to each node. Together this forms the node graph and is necessary data for a valid SMC result. Node n_a receives the IP address for node n_b . This allows n_a to know who to send the current result to. Additionally it allows the coordinator to specify a random order in which to perform the SMC and re-shuffle it for every operation. Importantly, the message that is sent from each node to the next must also contain the current result value up to this point to make this work. This is not shown in Figure 4.1 for simplicity, but will be showcased later in other figures under section *Security requirements*.

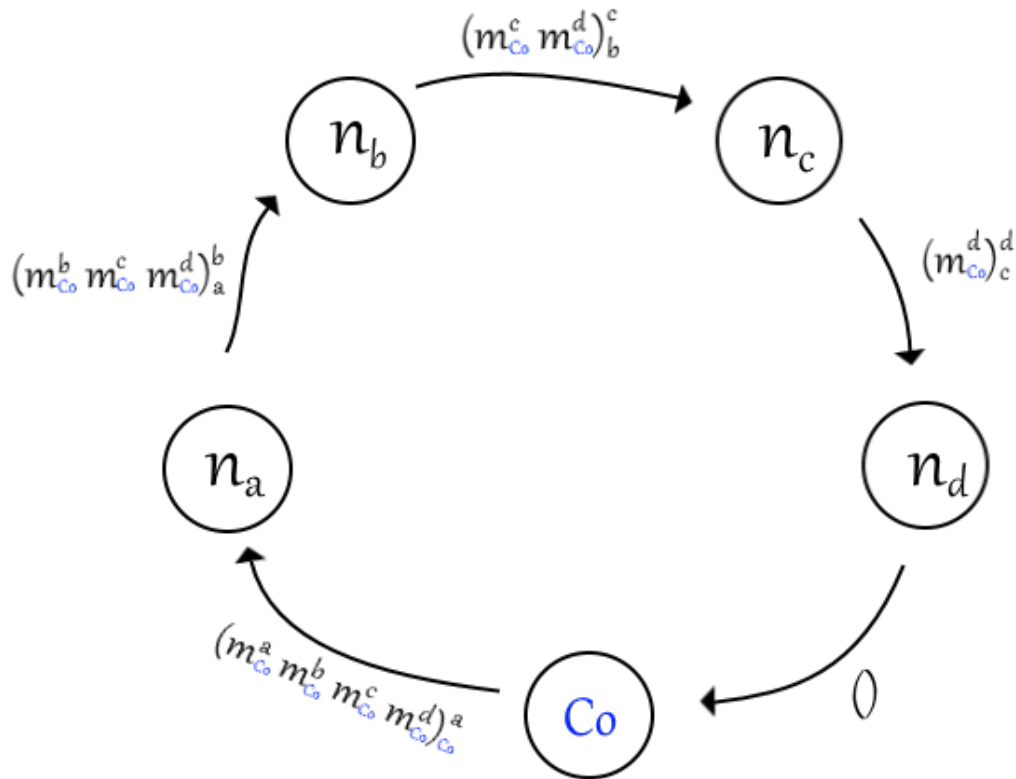


Figure 4.1: Node graph illustration

The result value is sensitive data and must be hidden from each node securely, while still being able to add the results together after each succeeding analysis step. Achieving this is also shown under *Security requirements*.

4.2 Security requirements

To show the security requirements needed to ensure a valid and secure SMC operation takes place. There will be displayed potential issues that can arise and subsequently explained why it happens and a possible way to fix it. A series of figures is shown with the same layout consisting of 5 nodes; Coordinator, Node-a, Node-b, Node-c and Node-d.

Starting with the most basic issue, where no data is encrypted. It's entirely built on trust. The network is extremely prone to eavesdroppers and frail to a large number of attacks. Any node can essentially pick up network packets,

alter them easily (as no encryption is used) and deliver the packets to the intended receivers. This issue is shown in figures 4.2 - 4.4 where it uses the node graph as an example. It displays why it's so important to partially encrypt the node graph. Otherwise, each node is able to alter the SMC order specified by the coordinator. One dirty node is all it takes to receive the analysis result of another node, which discloses a node's sensitive information. An example of how it could be achieved is if n_a alters the *send_list* by injecting its IP, such that n_b sends its result back to n_a , shown in figure 4.3. Another example is to simply eavesdrop the result shown in figure 4.4.

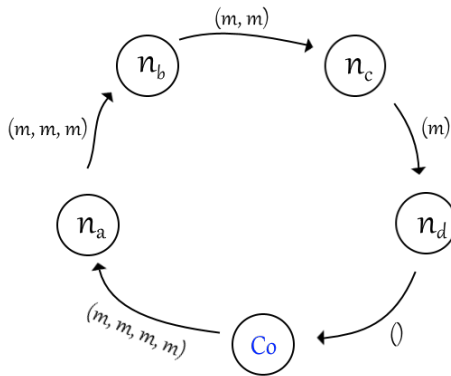


Figure 4.2: No encryption

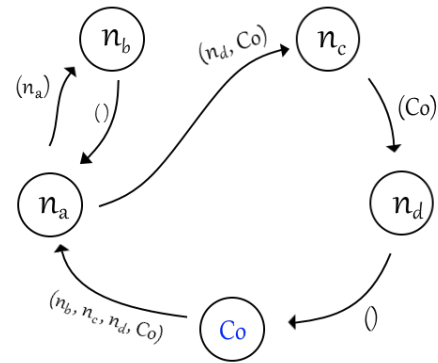


Figure 4.3: N_a injection

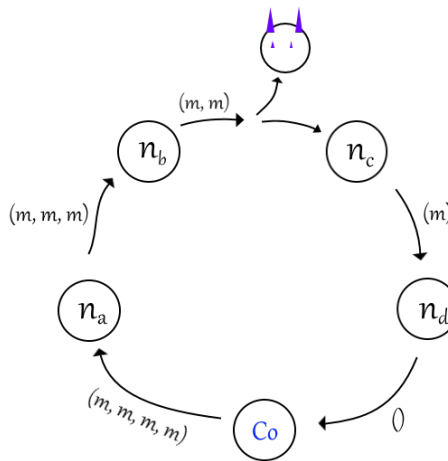


Figure 4.4: Eavesdropper listening

Evidently, as can be seen, having no encryption does not work well. It must be present in the implementation of an SMC algorithm. Figure 4.5 displays the *send_list* partially encrypted as well as the result values encrypted between each step. The encryption data is however not signed by the coordinator or by any nodes when data is being sent from one node to the next. Having encryption prevents any eavesdroppers to gain restricted knowledge as was previously possible. The most obvious unintentional information disclosure has been fixed, a step in the right direction. However, the SMC network is again very frail and all it takes is one dirty node to have information disclosure.

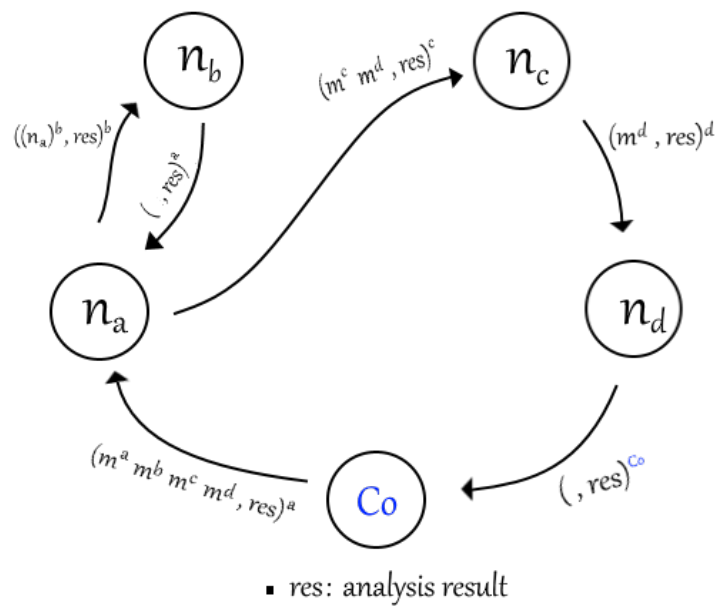


Figure 4.5: n_a injection, encryption but no signature used

Suppose n_a is again dirty and wants access to n_b 's analysis results. It should be illegal to gain this information. n_a can't alter the message to n_b from coordinator, as it has been encrypted to n_b only. What it can do however is remove the message altogether, recreate it and re-encrypt it to n_b 's public key. Since signing data is not used in this scenario, n_b has no way of knowing who the author of the message is and thus can't ensure the validity of the message. n_b will assume it's encrypted from the coordinator (which it originally was) and will send its result values to the received decrypted IP address from this message. n_b doesn't know that n_a changed it, such that n_b sends the result value back to n_a . n_a has managed to trick the system to gain private information not intended for it to know, by injection itself in the node graph.

As was mentioned in the *Concepts* section of this paper, signing data allows for data integrity. By signing with a private key, anyone can validate that the

original author is who he says he is, by verifying it with their public key. As figure 4.5 shows, signing each message is a necessary requirement to prevent any information disclosure by injection and dirty inside nodes.

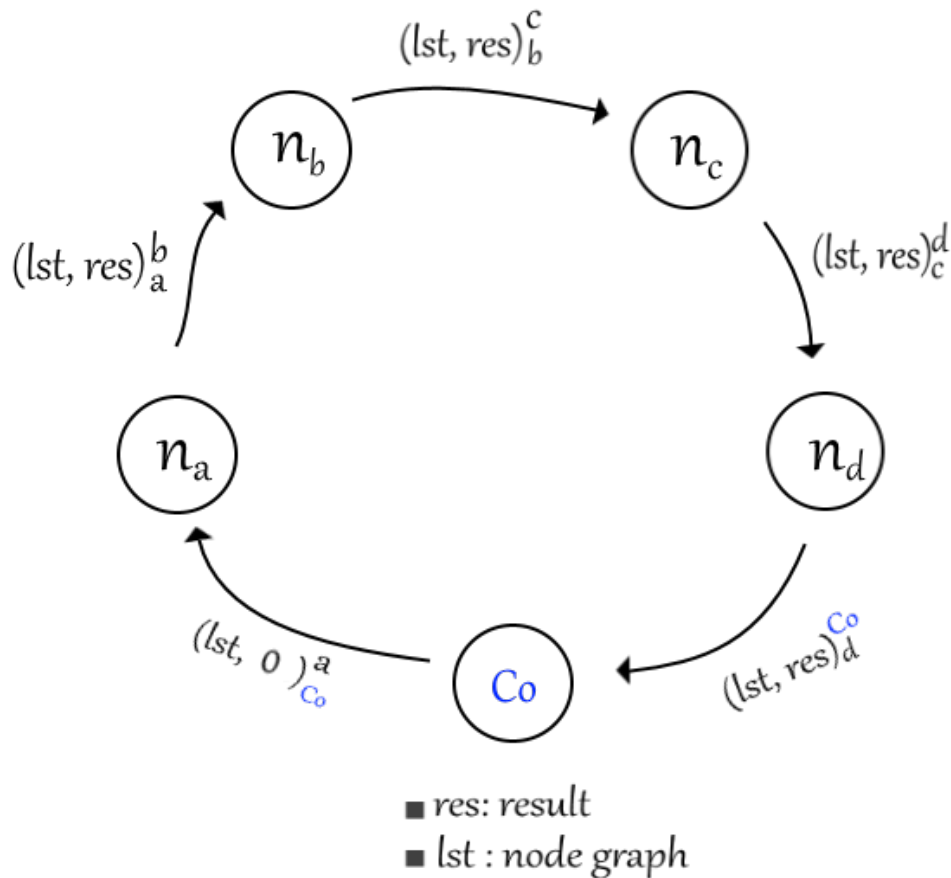


Figure 4.6: SMC: Signing data

Signing data is critical for a valid SMC operation. Figure 4.6 displays data encryption as well as data signing between each node. To make the figure easier to read, *lst* is denoted as the node graph, identical to the one shown in figure 4.1. *lst* therefore contains the entire partially encrypted and signed node graph. n_a is no longer able to inject itself into the node graph to confuse n_b . This is because n_b will now validate each message it receives to make sure it's from the correct sender, and that data has not been altered during transmission. Information disclosure by injection is no longer possible.

Still, the algorithm is not complete. There are two subtle yet critical issues remaining. As shown in figure 4.6, the Coordinator sends a result of zero to

n_a . No node has added their results to the sum result yet. n_a will perform its analysis operation and add its result value to the result sum. It pops n_b 's address from the partially encrypted node graph and furthers everything to n_b . The issue lies here, n_b will receive n_a 's result value. n_b must be able to decrypt this result as it will add its own result value to it, but this discloses n_a 's information. The premise of our SMC algorithm has been broken. The individual analysis result is sensitive information. The result can only be viewed if enough nodes has added their result together, thereby obscuring the result for the individual value.

Issue number two is the opposite of injecting data into the node graph. Every node can in theory delete information from it. Additionally, every node can set the result to any value they prefer. If node n_c sets the result sum to zero and furthers it to the last node n_d . n_d will add and send its individual result to coordinator. Only n_c will know that the entire result consist of only n_d 's values. If coordinator then makes the result public, then n_c can request the data and n_d 's information has been disclosed.

This is the reason for the circular structure of the SMC algorithm. It's because the Coordinator must initialize the result value to some random data, and at the end subtract it when it received the result back. This process is referred to as the round-robin [4] scramble in this thesis. To prevent information disclosure by furthering data to the next node, the result value is scrambled for all nodes except the coordinator. Coordinator starts of the SMC algorithm and sets the result to be equal to some random scramble value. It furthers the request to n_a who performs its own analysis and adds it to the result. When n_a furthers the request to n_b , n_b will not know n_a 's result value anymore. The scramble value has hidden it. When coordinator then receives the result back from n_d , it can remove its own scramble data and be left with the total analysis result from all nodes participating. Each individual result value can't be interpreted from the total result either, as long as there is enough nodes participating.

Concerning issue number two as previously stated, if n_c deletes information by setting the result to zero and sends it to n_d . Then n_d 's information will only be disclosed to the coordinator node, as n_d will further the data back to coordinator. Additionally, coordinator does not realize that n_d 's information was disclosed as it assumes its scramble value is still present. Coordinator removes its scramble and the end result will be wrong. n_c will not be able to retrieve n_d 's information. Scrambling the data will therefore prevent information disclosure by deleting node graph elements. As well as prevent the second node in the node graph to see the first nodes information.

The figure is updated in 4.7 to apply a scramble. It will not be possible for any clean node to understand the previous node's data. It's impossible to know

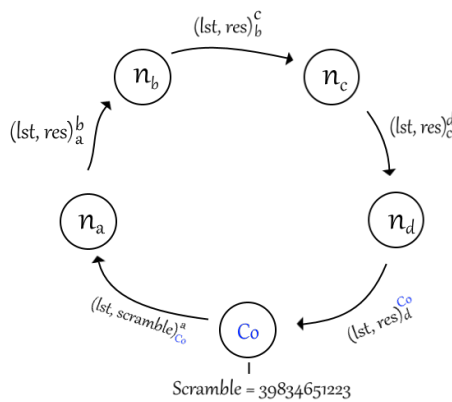


Figure 4.7: SMC: With scramble

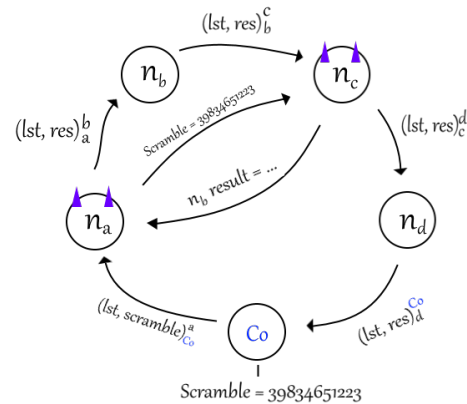


Figure 4.8: SMC: Scramble issue

what's real and what's scrambled between each analysis step. Node n_b can therefore not know n_a 's result anymore. However, if two dirty nodes communicate, it is possible for them to figure out the coordinator's scramble, thus able to disclose a nodes hidden information. As illustrated in figure 4.8, n_a and n_c are dirty nodes (represented by their evil purple horns), n_a receives the initial result value from the coordinator node. This value will simply be the scramble value. n_a communicates the scramble value with n_c , such that n_c knows what the secret scramble is. n_a then continues the SMC operation, adds zero to the result and furthers it to n_b . The clueless node n_b performs its analysis and adds it to the result and furthers the operation to the dirty node n_c . Since n_c already received the coordinator's scramble value from n_a , it can subtract it from the sum result from n_b , and its information is disclosed.

Having one scramble value is not sufficient enough to satisfy our SMC premise since SMC shouldn't assume having all clean inside nodes. Therefore there must be an unique scramble added to the result between each node. Each value is then protected by the scramble such that the issue where two nodes working together to disclose information, will not be possible anymore. Inherently, it will also properly prevent information disclosure by deleting data (setting result equal to zero). Figure 4.9 shows the structure with unique scrambles for each node. There are two possible ways of designing this structure in a practical case. Either coordinator generates all the scrambles and encrypts them to their corresponding recipient. Or each node generates their own scramble value and encrypts it to the coordinator. Coordinator must either way know all of the scramble values to be able to subtract it from the result sum. Therefore it does not matter which design solution is used in terms of preventing information disclosure, both are functionally the same. Additionally, it does not matter in terms of network bandwidth nor for simplicity. This design chose to generate all scrambles on coordinator, and send it encrypted alongside the other metadata

needed when sending SMC analysis requests.

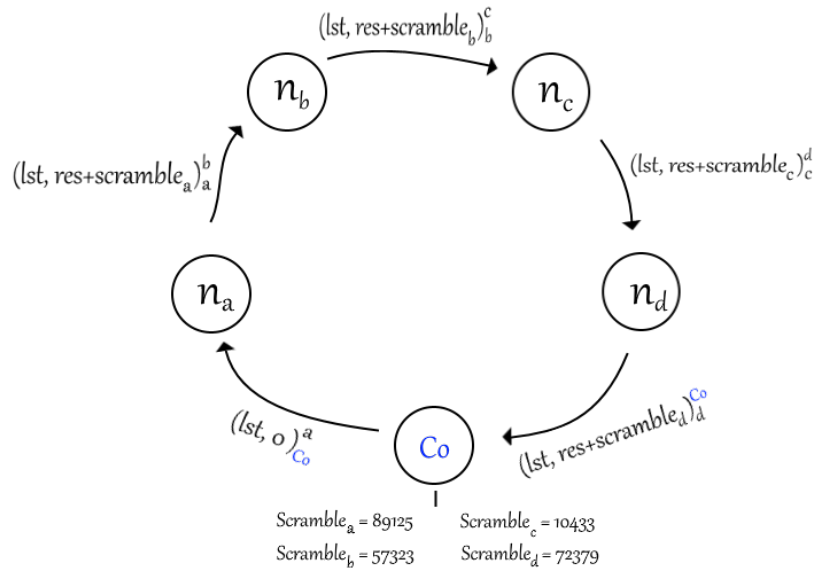


Figure 4.9: Unique scrambles

The SMC algorithm can potentially disclose information again. Even after utilizing unique scrambles for each node. When all nodes in the network are dirty except one and the coordinator, it will be possible for them to leak information. Assume n_a , n_c and n_d are all dirty nodes and they wish to retrieve n_b 's analysis result. Since the dirty nodes communicate with each other, they can share their results among themselves. Otherwise, and for simplicity of this example, they can agree to add zero to the result. n_a adds zero, furthers the request to n_b who adds its true result. n_b then furthers to n_c who adds zero to the result and furthers it to n_d . n_d does the same thing, adds zero and lastly sends it back to the coordinator. Eventually, what coordinator will receive is this.

$$\begin{aligned}
 result &= (0 + s_a) + (m_b + s_b) + (0 + s_c) + (0 + s_d) \\
 result &= \cancel{s_a} + m_b + \cancel{s_b} + \cancel{s_c} + \cancel{s_d} \\
 result &= m_b
 \end{aligned}$$

All nodes added their unique scramble values such that when coordinator removes the scrambles, it is left with only n_b 's result value. n_b 's individual result has therefore been disclosed. Without necessarily knowing it, coordinator has received only the result for n_b . Additionally, if coordinator makes this result publicly available then all of the other dirty nodes knows n_b 's private data. Thus the SMC algorithm has been broken and is no longer reliable for further use with the same sequence of nodes. Sadly, there is no attainable way to catch nor

prevent inside collaboration of nodes in this way. No matter what measures is done to prevent information leak, if there is full inside collaboration, and if coordinator makes the result public, data will always be as secure as the nodes who produced it. The only way to guarantee an ideal SMC operation is if a certain amount of nodes can be guaranteed clean. This guarantee is impossible for all practical cases as has been showed, because nodes can always communicate outside the SMC network. Meaning that it's outside the scope of what the SMC algorithm can control. Figure 4.10 shows an example.

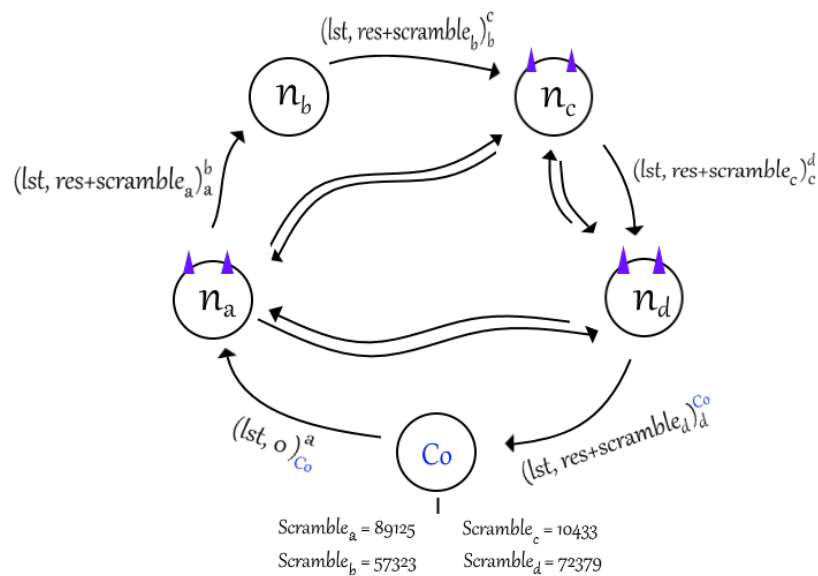


Figure 4.10: Unique scrambles issue

Lastly, we have always assumed a clean coordinator node, but what happens if the coordinator is dirty? In this case, information from any node is very easily disclosed. All nodes trust the coordinator by default and if coordinator asks for some result value, it will receive it. Coordinator can fake SMC operations. It is therefore of significant importance that coordinator must be a clean and a valid leader node in the network. The whole algorithm relies on it, which is why the coordinator must have a valid certificate signed by a trusted authority. For each request that is sent, the certificate must be properly checked to ensure clean nodes, not only for coordinator but for all other nodes in the network as well. If nodes are who they claim to be, then the SMC algorithm will be as safe as the encryption algorithms used within. If Nodes certificates has been signed by a trustworthy authority then this is as safe as it gets. The baseline for the security measures in this paper is therefore set with the certificates such that each node can be trusted to not be dirty, and thus won't communicate with other nodes in the network to gain illegal knowledge of other nodes. The only way to disclose information would be for inside collaboration for all but

one node or if coordinator is dirty. Theoretically possible, but impractical to assume.

4.3 Minimum nodes needed

Once a valid SMC operation has taken place, at the end of it, coordinator receives the result sum and because of the size of the network (i.e. 5 nodes in the network), it won't know the individual results. It won't be able to deduce private information as each individual result added obscures the result sum further. To keep a simple and uniform design, the minimum number of nodes in a network must be supplied from a use case perspective. Assume two users, hospital *A* and hospital *B*. They wish to perform Pearson's R coefficient on their joint data, to see if there are any correlation between their patients health records and an arbitrary disease. *A* contains unknown sick patients and *B* contains known patients with the disease.

By looking at the correlation, it is possible to predict how likely someone from *A* has the disease. In this case, the minimum number of nodes in the SMC network must be 2 to allow it to pass through. If this is accepted by all three nodes in the network (coordinator, *A* and *B*), then the operation goes forward. In this special case, *A* is the only node who needs to know the end result. *B* does not need to know, as it was only used to help *A* decide if their patients had the disease. Therefore, to make it safer for *A*, coordinator can be run in parallel on node *A*. When coordinator receives the result, it will be received on *A* and the result does not need to be made public, *B* will therefore not know the end result.

As has been shown, minimum number of nodes in the SMC network is special case and may vary immensely from different situations. With one node in the network, coordinator will receive the immediate result of this one node. In most cases this is a security breach. It can however be allowed by user if minimum nodes is set to 1. This is a special case where the analysis result is not deemed sensitive information by the user, therefore this data can be freely distributed.

4.4 Sneak communication

The server uses a threading variant of the *BaseHTTPRequestHandler* class from the *http.server* module in Python [20, 21]. This allows handling of multiple requests near simultaneously. Although Python does not support true thread-

ing since it's limited by its global interpreter lock (GIL), it works well for I/O operations such as handling server requests [22].

When the server starts it sits idle while waiting for requests. If the server is coordinator, it must be specified in command line argument when starting and also be given a whitelist (nodefile) of all nodes in the system. Examples of this can be found in Sneak's documentation in the appendix section. The coordinator must also be started last and the reason is because it will send a get request for the certificates of all the other servers specified in its network.

The background server natively uses the Sneak cryptography module for each input and output message request. Which means it operates closely with our own module. Every time a new message is sent/received it is safely encrypted/decrypted. All transmission data is signed and the certificate is properly checked. Otherwise the server itself functions like any other HTTP server. It uses RESTful API as the architectural style for handling requests. The distributed network of servers is centralized with the coordinator and waits for "*GET /analysis*" requests.

4.5 Cryptography module

This module consist of a single class called *Crypt* and also has a test to validate its correctness. The test is under Sneak's source code, which can be found when pip installing Sneak.

The cryptography submodule is meant to be abstracted away from the end-user completely. However, It's possible to use it to create any custom implementations, although this is not needed while using the SMC module. The *sneaksmc.client* submodule uses the cryptography submodule to communicates with the coordinator node for the user.

The cryptography submodule is fairly simple to use for any end-user as they only have two functions to use, namely encrypt and decrypt. As the name suggests, encrypt will take a message and encrypt it securely to any public key given. The return value will be a Python array of bytes (not to be confused with bytestring). In the background quite a lot more has been done and the return data contains metadata needed for efficiency and simplicity. Needed metadata includes senders public key, public key size in bytes, initialization vector(IV), signature (encrypted hash value), encrypted symmetric key and lastly the encrypted message. The metadata is handled within the encrypt and decrypt functions.

Since symmetric encryption uses block ciphering, the message input has to be a byte length multiple of 16. Naturally, the end-user doesn't need to handle this but the message must be padded in the encrypt function to its nearest multiple. Therefore, the padded message character cannot be used in the original message, otherwise undefined behaviour will occur. The null terminator character `\0` is used for this purpose. An example of how to use this cryptography submodule is shown below in Listing-4.1. Note this does not use the background server to send messages over HTTP requests. Usually Alice and Bob would be on two different nodes.

```
1 from sneaksmc.crypt import Crypt
2
3 bob = Crypt()
4 alice = Crypt()
5
6 message = "Hi Alice, this is bob"
7
8 # bob sends to alice
9 encrypted_blob = bob.encrypt(alice.get_public_key(),
10                             message)
11
12 # Alice decrypts
13 msg = alice.decrypt(encrypted_blob)
14
15 print(msg)
16 # ~ Should print "Hi Alice, this is bob"
```

Listing 4.1: Using the Sneak cryptography module

4.6 Setting up servers

Starting the server requires little effort for regular nodes that isn't coordinator. The SMC background server is imported into a Python script, the `sneaksmc.server` submodule has a function called `run_server()` which takes in a number of parameters. For regular nodes, only the parameter `analysis_function` and `coordinator_addr` is strictly needed.

Since the complexity and functionality of the background server is abstracted away as much as possible from users. The analysis function and coordinator address holds crucial information needed during runtime. For the analysis function, this reference must be given during initialization. The reason for this is that all nodes in the network might store their sensitive data differently, or might perform analysis on it differently. Meaning it will be hard to have one predefined function to handle all use cases for all nodes in the system. The `coordinator_addr` must similarly also be specified.

An example of how to create the Python file and start the background server is shown in Listing 4.2. The server must be run manually by the user by typing, "python3 example.py" which will run the background server and use the analysis function defined in this file.

```

1   from sneaksmc import server as sneakserver
2
3   def analysis(data_in):
4       # Perform analysis needed on its sensitive data.
5       # How this is done is up to end user.
6       # This example retrieves data from database,
7       # and adds it to the sum value.
8       db_connect = db.connect(db_connection, db_password)
9       sensitive_data = db_connect.get("sensitive_data")
10
11      assert type(data_in) is int
12      data_out = data_in + sensitive_data
13
14      # Must return same format as argument given.
15      assert type(data_out) is int
16      return data_out
17
18      if __name__ == "__main__":
19          # Runs server on this thread but can be spawned
20          # on a new thread here if needed.
21          sneakserver.run_server(analysis_function=analysis,
22                                coordinator_addr="localhost:8080")
23
24

```

Listing 4.2: Using the Sneak SMC server

When coordinator starts up, it must know about all other nodes in the system. This is achieved by using a nodefile, which is a list that contains all addresses (ip:port) of nodes allowed in the network. Therefore, to run the server as coordinator node, an additional parameter is needed for *run_server()*, which is *nodefile*. It's the location and filename for where this file exists. Listing 4.3 similarly shows how to set up the coordinator server. Note that coordinator does not need to be started from a custom script like the other "regular" servers who provide the analysis function. After pip installing the library, the coordinator can be run directly from the terminal. This goes for the client (*sneaksmc.client*) as well.

Alternatively running coordinator server is done by typing "python3 *sneaksmc/server.py -n nodes.txt*" in terminal. This way prevents the need for a custom python script and is therefore meant as the main way of running the coordinator.


```
1  from sneaksmc import server as sneakserver
2
3  # Coordinator does not need analysis function
4  # if it's not part of the smc analysis operation part.
5
6  if __name__ == "__main__":
7      sneakserver.run_server(nodefile="nodes.txt",
8                             min_nodes=4,
9                             auto_shutdown=60*15) # 15 min
10
11
12
```

Listing 4.3: Using the Sneak SMC server as coordinator

Coordinator does not need to define *coordinator_addr* as this will be defined by the *nodefile* argument. Additionally, it does not need to be part of the analysis operations, thus not need an analysis function argument to be defined. Once the coordinator is set up, the nodefile will be broadcast to each of the nodes contained in it. Allowing everyone to know about every other node, such that messages can be validated and properly encrypted to their public keys upon communication in an SMC operation.

Optionally, *min_nodes* can be set as well which decides the minimum number of nodes the network must have to perform a valid SMC operation. This can vary immensely for different use cases and is therefore best left open for users to decide. The last optional parameter is *auto_shutdown* which decides when the server should automatically shut down. This is time given in seconds and prevents unnecessary use of resources if it's left running idle.

Leveraging this structure makes it possible to perform complex analysis in any way the user needs. It is also convenient and easy to integrate into other code as it's segregated, and then shut the server down whenever it isn't needed. The user can in theory build an automatic system to perform their SMC operations for one time or continuous use. The server, cryptography and client module is all segregated and can be used independently.

4.7 Shutting down server

To shut down the server, the Sneak client module can be used to send a shutdown request to the coordinator. Subsequently, coordinator will further this shutdown request to all other nodes in the system, causing every node to shut down gracefully. Listing 4.4 shows an example use.

```

1  from sneaksmc import client as sneakclient
2
3  if __name__ == "__main__":
4      coordinator_addr = "196.112.90.35"
5      sneakclient.shutdown_smc(coordinator_addr)
6
7

```

Listing 4.4: Using the Sneak client module

To shut down specific nodes can also be done either on local machine or remote by using the Sneak client. Sending a shutdown request to any node except coordinator, will shut down that node specifically.

4.8 Running SMC operations

A client is the one starting SMC operations and any node can be a client. It means that the *sneaksmc.client* submodule is used to send a "start_analysis" request to the coordinator. There are two ways of sending this request. The client can be run directly in terminal after having installed the library, i.e. "*python3 sneaksmc/client.py -coordinator localhost:8089*". Otherwise the *sneaksmc.client* submodule can be imported to any Python script and started from there. Listing 4.5 shows an example.

```

1  from sneaksmc.client import Client
2
3  if __name__ == "__main__":
4      coordinator = "localhost:8089"
5      c = Client()
6      code, id = c.request_analysis(coordinator)
7
8      if code == 200:
9          code, res = c.get_result(id)
10         print("Result is " + res)
11     else:
12         # Error requesting analysis
13         pass

```

Listing 4.5: Using the client submodule to start a SMC operation and retrieving the result.

Once a "/start_analysis" request is received in Sneak, the operation is initialized and begun. Firstly, if the server receiving the request is not coordinator then an error is raised and the request is simply dropped. Additionally, if the client sending the request does not have a valid certificate, nor is in the coordinator's whitelist of nodes, then the request is similarly dropped. This ensures that the coordinator has full control over all operations.

Next, coordinator initializes a dictionary which contains all the relevant data being transmitted. The dictionary has mainly two key-value pairs, "send_list" and "data". The "send_list" is the node graph and the scramble data needed by each node. This will be showcased in Listing 4.6 below. To reiterate, the node graph is a Python list of node's IP addresses (using *ip:port* format) and is the overall sequence of nodes to perform their analysis tasks. This sequence is randomized for each operation that takes place. The first node n_a will not be in the "send_list" as the coordinator knows to send it there (to n_a). Node n_b will therefore be first, and n_a can pop n_b 's IP address from the list and further the operation there once its done.

The "data" element in the dictionary holds the result produced by nodes, in addition to their scramble data to make it unreadable. Each IP address in the "send_list" is encrypted to a node's public key and therefore only that node can decrypt the next address in order to progress the operation. This is a partial encryption design to prevent unnecessary encryption if comparing to layered encryption. The code snippet below (Listing 4.6) shows the code for how the coordinator sets up the analysis operation to be performed across all nodes in the distributed system. It's important to note that the coordinator node is the last node in the "send_list" and is therefore the last node to (potentially) perform the analysis operation. This means that the coordinator node will have the end result of all the analysis data that has been done on other nodes. The coordinator is the one with the random scramble values that obscured the result ("data") and can therefore subtract all scrambles to obtain the true result of the SMC analysis securely. It's also important that the coordinator is the one storing the result as the client can then send a get request for this end result once the operation is done.

```

1 elif self.path.startswith("/start_analysis"):
2     global scramble_list
3
4     is_valid, msg = self.validate_request()
5     if not is_valid:
6         send_response(404, "Request not valid: "+msg)
7         return
8
9     send_list = []
10    dictionary = {'send_list': send_list, 'data': 0}
11
12    # Read node IPs + public key from file
13    node_list = []
14    with open(certificate_file, "r") as f:
15        for line in f.readlines():
16            addr, strkey = line.split("@")
17            node_list.append((addr, cryption.
18                            string2key(strkey)))
19

```

```

20     # Shuffle the node list to make the order random
21     random.shuffle(node_list)
22
23     # Partially encrypt the send list
24     for i in range(len(node_list)):
25         addr, key = node_list[i]
26
27         # Last node sends back to coordinator
28         if i >= len(node_list)-1:
29             co = cryption.encrypt(key, coordinator_address)
30             scramble = get_scramble()
31             scramble_list.append(scramble)
32             msg = {'next': co, 'scramble': scramble}
33             msg = json.dumps(msg)
34             send_list.append(msg.decode('latin-1'))
35             break
36
37         sendto_ip, _ = node_list[i+1]
38         scramble = get_scramble()
39         scramble_list.append(scramble)
40         msg = {'next': sendto_ip, 'scramble': scramble}
41         msg = json.dumps(msg)
42         msg = cryption.encrypt(key, msg)
43         send_list.append(msg.decode('latin-1'))
44
45     # Dump the dictionary to string such that it can be sent
46     over http
47     str_dictionary = json.dumps(dictionary)
48
49     # Send the string dictionary to the first node in the list.
50     first_node, first_node_pkey = node_list[0]
51
52     enc_dictionary = cryption.encrypt(first_node_pkey,
53                                     str_dictionary)
54
55     status, content = client.send_client_request(type="POST",
56                                                url="/analysis",
57                                                body_=enc_dictionary,
58                                                receiver=first_node)
59
60     if status != 200:
61         send_response(404, "Error occurred: %s" % content)
62         return
63
64     # Send ok response back to client.
65     send_response(200, "ok")

```

Listing 4.6: Start analysis snippet

A way to improving the scalability of this design is to make it compatible with concurrent requests. Each SMC operation must have a unique ID such that when coordinator receives the last "analysis" request, it can store the result by mapping it with the ID. A client who request to start an analysis operation will

receive the ID of the operation back from the coordinator as a reply. The client can then request the result by specifying the operation ID. This can be seen in Listing 4.5 above.

/5

Examples and experiments

5.1 The fair competition problem

To show an example of the library and some of its use cases, we first look at the fair competition problem and how it can be solved practically with Sneak. We already know that it can't be solved with only two judges as it's impossible while still maintaining the anonymous votes. Instead there will be three judges, Alice, Bob and Charlie. Each of the three judges must set up their server and provide their vote function (to figure out if they vote yes or no). Listing 5.1 shows how Alice, Bob and Charlie sets up the server. Note that the vote function uses `input()` which blocks and waits for user input. If they write `1`, this means yes and `0` means no. This way they are able to run multiple SMC operations without having to restart the server.

```
1 import sneaksmc.server as sneak
2
3 def vote(sum_votes):
4     result = input()
5     while (result != "0" and result != "1"):
6         print("Vote must be either 0 or 1")
7         result = input()
8
9     print("You voted %s" % ("yes" if result == "1" else "no"))
10
11     sum_votes += int(result)
12     return sum_votes
13
14 if __name__ == "__main__":
```

```
15 caddr = "localhost:8089"  
16 sneak.run_server(analysis_function=vote, coordinator=caddr)
```

Listing 5.1: Alice Bob and Charlie starting server

Figures 5.1 - 5.4 shows what it looks like after running the servers with the terminals up. Additionally, there must be a coordinator server running shown in figure 5.4.

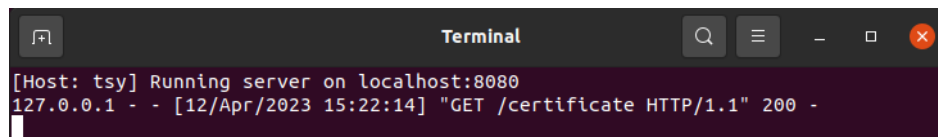


Figure 5.1: Alice's server

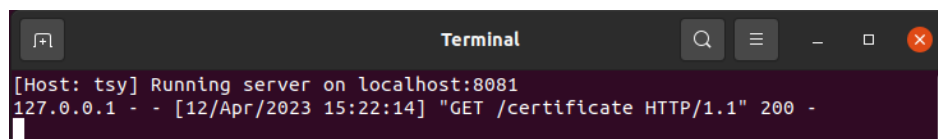


Figure 5.2: Bob's server

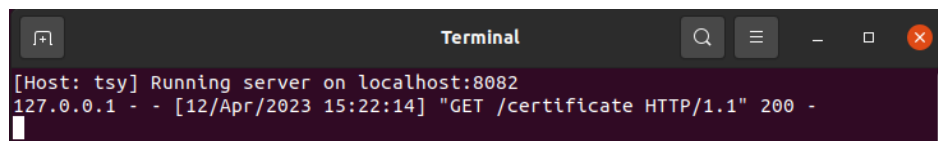


Figure 5.3: Charlie's server


```

localhost:8080@12388231793698076117692315929605235298618238514049654371019864658
61061005465841605324987547853811874045078241185549471179327824355223370056491322
33220851173288779368233958550978701042585759876404090137240119124444048706381094
25912989339741662621047278108262034273593069545075501032137711801609053801703113
3469:~:65537
localhost:8081@10101408555919041372477408238812671070391351574938603317824011852
77331963061332879926267484920865350666959779211042588895419001891644096270701408
09014182796036922488718943671944473894398664538155683322067097022252172574129723
16280049495003202986637651590076575537069406400716149872375266261491485105377057
2921:~:65537
localhost:8082@14318513999459225596888620723378922963422085895307913576623225762
42252370064116748519966258182227868415302123750018232720640000271963810529001011
44061426980139496023671222973896638139802015794045567932416200367139740845083943
28203770826850980552681249909183937911728568013093203961590976761447602819394226
3419:~:65537
[Host: tsy] Running server on localhost:8089

```

Figure 5.4: Coordinator's server

Coordinator's server has sent a GET request for the certificates of Alice, Bob and Charlie which can be seen on their servers ("*GET/certificate*"). These certificates is printed out on the coordinator's server for the convenience of this example. The servers are connected together and been fully initialized.

Now a contestant can send a request to start the vote. The script for the contestant is shown in listing 5.2.

```

1 from sneaksmc.client import Client
2
3 if __name__ == "__main__":
4     coordinator = "localhost:8089"
5     c = Client()
6     code, id = c.request_analysis(coordinator)
7     code, res = c.get_result(id)
8     print("Vote result: %s" % res)

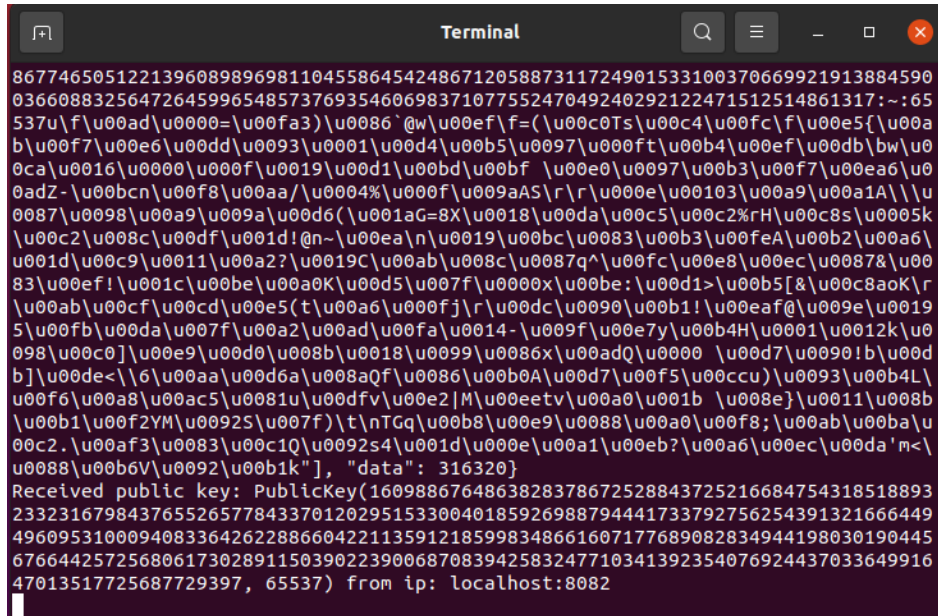
```

Listing 5.2: Contestant starting a vote operation and retrieving the result

Note that in order for the contestant to be able to start an SMC operation with Sneak, the contestant must either be known to the coordinator (whitelisted), or the SMC network is publicly available (public setting is set to true). If so then anyone can start an operation on this network and retrieve the result. From Listing 5.2 we can see that an ID is returned when requesting to perform an analysis operation. This ID is from the coordinator and must be used in order to retrieve the correct result.

Figures 5.5 - 5.8 shows each of the servers after having run the SMC operation to vote for the contestant. Figure 5.5 and 5.6 has quite a lot of unreadable text shown. The reason is to show the encrypted node graph (send_list) created by the coordinator server. It also shows the "data" component which holds the current SMC result. This value is seen completely different from each judge

server. Alice sees 316320, Bob sees 59923 and Charlie sees -3940. This is due to the scrambling done to protect their individual results. We can also see the data that is encrypted and the volume of it for only 3 participating judges. All of the encrypted data is the node graph (on Figure 5.5 and 5.6).

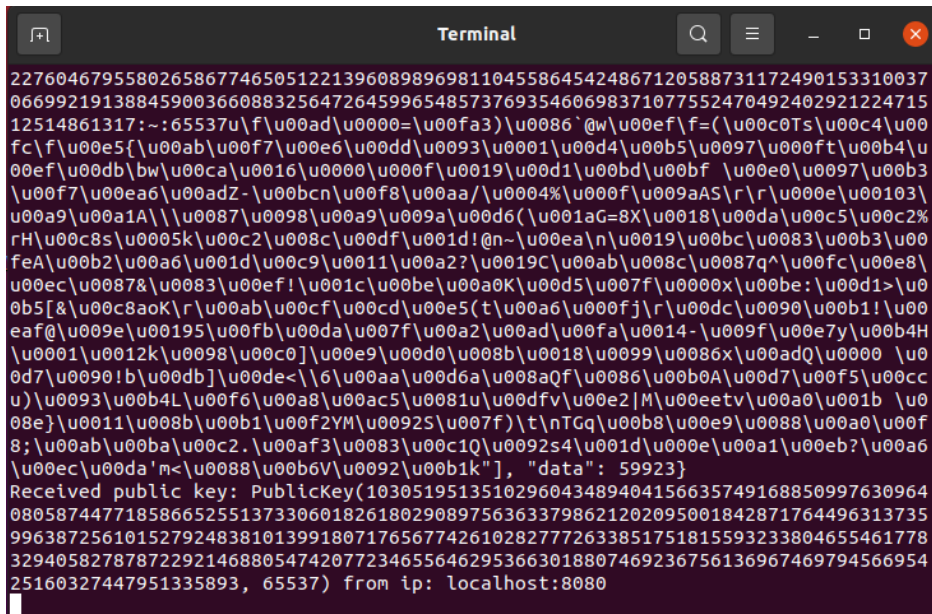


```

86774650512213960898969811045586454248671205887311724901533100370669921913884590
036608832564726459965485737693546069837107755247049240292122471512514861317:~:65
537u\|f\|u00ad\|u0000=\|u00fa3)\|u0086`@w\|u00ef\|f=(\|u00c0Ts\|u00c4\|u00fc\|f\|u00e5{\|u00a
b\|u00f7\|u00e6\|u00dd\|u0093\|u0001\|u00d4\|u00b5\|u0097\|u000ft\|u00b4\|u00ef\|u00db\|bw\|u0
0ca\|u0016\|u0000\|u000f\|u0019\|u00d1\|u00bd\|u00bf \|u00e0\|u0097\|u00b3\|u00f7\|u00ea6\|u0
0adZ-\|u00bcn\|u00f8\|u00aa\|u0004%\|u000f\|u009aAS\|r\|r\|u000e\|u00103\|u00a9\|u00a1A\|\|u
0087\|u0098\|u00a9\|u009a\|u00d6(\|u001aG=8X\|u0018\|u00da\|u00c5\|u00c2%rH\|u00c8s\|u0005k
\|u00c2\|u008c\|u00df\|u001d!@n-\|u00ea\|n\|u0019\|u00bc\|u0083\|u00b3\|u00feA\|u00b2\|u00a6\
u001d\|u00c9\|u0011\|u00a2?\|u0019C\|u00ab\|u008c\|u0087q^\|u00fc\|u00e8\|u00ec\|u0087&\|u00
83\|u00ef!\|u001c\|u00be\|u00a0K\|u00d5\|u007f\|u0000x\|u00be:\|u00d1>\|u00b5[\|u00c8aoK\|r
\|u00ab\|u00cf\|u00cd\|u00e5(t\|u00a6\|u000fj\|r\|u00dc\|u0090\|u00b1!\|u00eaf@\|u009e\|u0019
5\|u00fb\|u00da\|u007f\|u00a2\|u00ad\|u00fa\|u0014-\|u009f\|u00e7y\|u00b4H\|u0001\|u0012k\|u0
098\|u00c0]\|u00e9\|u00d0\|u008b\|u0018\|u0099\|u0086x\|u00adQ\|u0000 \|u00d7\|u0090!b\|u00d
b]\|u00de<\|6\|u00aa\|u00d6a\|u008aQf\|u0086\|u00b0A\|u00d7\|u00f5\|u00ccu)\|u0093\|u00b4L\
\|u00f6\|u00a8\|u00ac5\|u0081u\|u00dfv\|u00e2|M\|u00eetv\|u00a0\|u001b \|u008e}\|u0011\|u008b
\|u00b1\|u00f2YM\|u0092S\|u007f)\|t\|nTGq\|u00b8\|u00e9\|u0088\|u00a0\|u00f8;\|u00ab\|u00ba\|u
00c2.\|u00af3\|u0083\|u00c1Q\|u0092s4\|u001d\|u000e\|u00a1\|u00eb?\|u00a6\|u00ec\|u00da`m<
u0088\|u00b6v\|u0092\|u00b1k"}], "data": 316320}
Received public key: PublicKey(1609886764863828378672528843725216684754318518893
23323167984376552657784337012029515330040185926988794441733792756254391321666449
49609531000940833642622886604221135912185998348661607177689082834944198030190445
67664425725680617302891150390223900687083942583247710341392354076924437033649916
47013517725687729397, 65537) from ip: localhost:8082

```

Figure 5.5: Alice's server

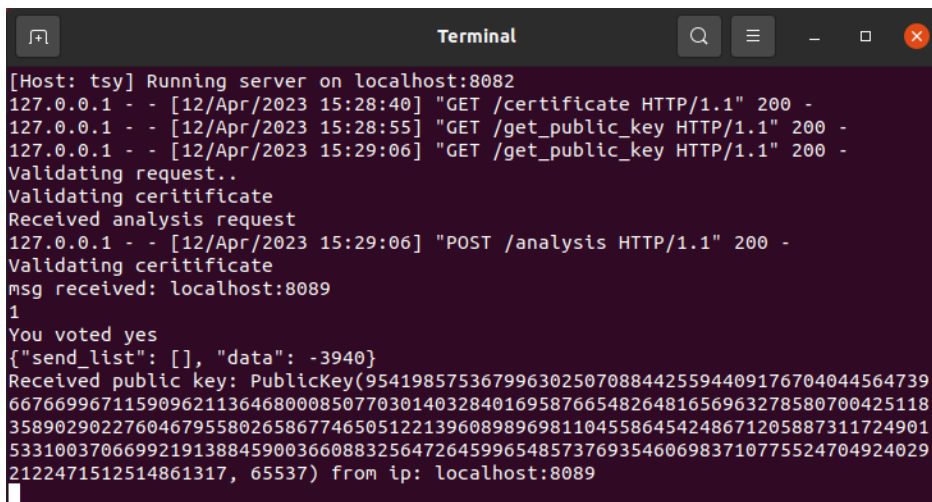


```

22760467955802658677465051221396089896981104558645424867120588731172490153310037
06699219138845900366088325647264599654857376935460698371077552470492402921224715
12514861317:~:65537u\f\u00ad\u0000=\u00fa3)\u0086`@w\u00ef\f=(\u00c0Ts\u00c4\u00
fc\f\u00e5{\u00ab\u00f7\u00e6\u00dd\u0093\u0001\u00d4\u00b5\u0097\u000ft\u00b4\u0
00ef\u00db\bw\u00ca\u0016\u0000\u000f\u0019\u00d1\u00bd\u00bf \u00e0\u0097\u00b3
\u00f7\u00ea6\u00adZ-\u00bcn\u00f8\u00aa/\u0004%\u000f\u009aAS\r\r\u000e\u00103\
\u00a9\u00a1A)\u0087\u0098\u00a9\u009a\u00d6(\u001aG=8X\u0018\u00da\u00c5\u00c2%
rH\u00c8s\u0005k\u00c2\u008c\u00df\u001d!@n~\u00ea\n\u0019\u00bc\u0083\u00b3\u00
feA\u00b2\u00a6\u001d\u00c9\u0011\u00a2?\u0019C\u00ab\u008c\u0087q^\u00fc\u00e8\
\u00ec\u0087&\u0083\u00ef!\u001c\u00be\u00a0K\u00d5\u007f\u0000x\u00be:\u00d1>\u0
0b5[&\u00c8aok\r\u00ab\u00cf\u00cd\u00e5(t\u00a6\u000fj\r\u00dc\u0090\u00b1!\u00
eaf@\u009e\u00195\u00fb\u00da\u007f\u00a2\u00ad\u00fa\u0014-\u009f\u00e7y\u00b4H
\u0001\u0012k\u0098\u00c0]\u00e9\u00d0\u008b\u0018\u0099\u0086x\u00adQ\u0000 \u0
0d7\u0090!b\u00db]\u00de<\\6\u00aa\u00d6a\u008aQf\u0086\u00b0A\u00d7\u00f5\u00cc
u)\u0093\u00b4L\u00f6\u00a8\u00ac5\u0081u\u00dfv\u00e2|M\u00eetv\u00a0\u001b \u0
08e}\u0011\u008b\u00b1\u00f2YM\u0092S\u007f)\t\nTGq\u00b8\u00e9\u0088\u00a0\u00f
8;\u00ab\u00ba\u00c2.\u00af3\u0083\u00c1Q\u0092s4\u001d\u000e\u00a1\u00eb? \u00a6
\u00ec\u00da`m<\u0088\u00b6v\u0092\u00b1k"] , "data": 59923}
Received public key: PublicKey(1030519513510296043489404156635749168850997630964
08058744771858665255137330601826180290897563633798621202095001842871764496313735
9963872561015279248381013991807176567742610282772633851751815593233804655461778
3294058278782292146880547420772346556462953663018807469236756136967469794566954
25160327447951335893, 65537) from ip: localhost:8080

```

Figure 5.6: Bob's server



```

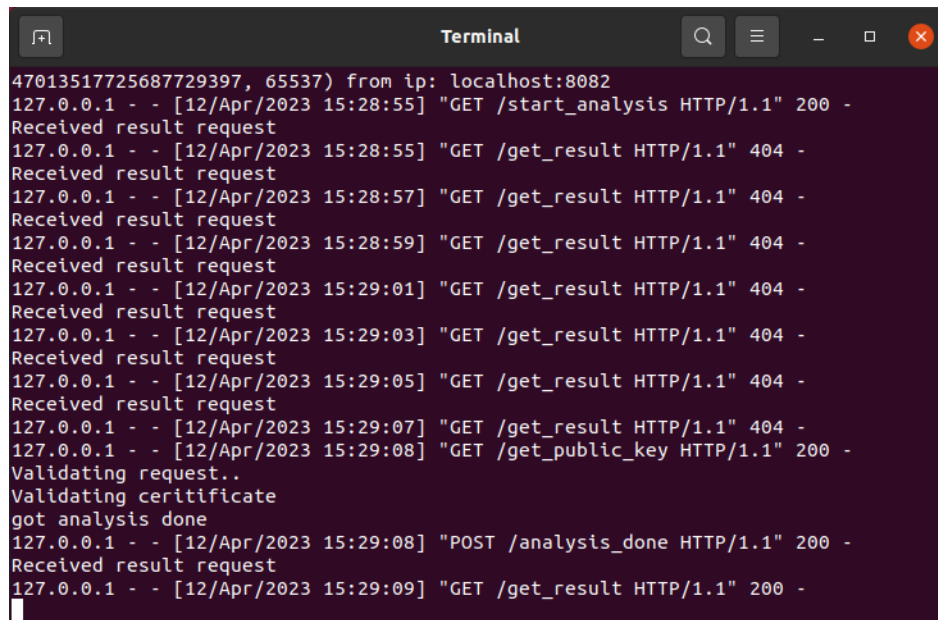
[Host: tsy] Running server on localhost:8082
127.0.0.1 - - [12/Apr/2023 15:28:40] "GET /certificate HTTP/1.1" 200 -
127.0.0.1 - - [12/Apr/2023 15:28:55] "GET /get_public_key HTTP/1.1" 200 -
127.0.0.1 - - [12/Apr/2023 15:29:06] "GET /get_public_key HTTP/1.1" 200 -
Validating request..
Validating certificate
Received analysis request
127.0.0.1 - - [12/Apr/2023 15:29:06] "POST /analysis HTTP/1.1" 200 -
Validating certificate
msg received: localhost:8089
1
You voted yes
{"send_list": [], "data": -3940}
Received public key: PublicKey(9541985753679963025070884425594409176704044564739
66766996711590962113646800085077030140328401695876654826481656963278580700425118
35890290227604679558026586774650512213960898969811045586454248671205887311724901
53310037066992191388459003660883256472645996548573769354606983710775524704924029
2122471512514861317, 65537) from ip: localhost:8089

```

Figure 5.7: Charlie's server

Charlie's server (figure 5.7) does not have all the unreadable text in the terminal, so it's possible to see the steps done from start to finish. It firstly receives the "GET /certificate" request from coordinator. Then coordinator sends a "GET /get_public_key" request in order to partially encrypt the node graph. The next "GET /get_public_key" is likely from Bob who finished his vote and wants to further the operation to Charlie, but must have his public key to encrypt the message. Once Charlie then receives the "POST /analysis" request, he will

validate the request and validate the certificate received. The second "validating certificate" which can be seen is Charlie validating the coordinator's certificate from the encrypted node graph. It will make sure that Charlie furthers the data to the correct IP address given to him. After validation, the vote function is performed, and we can see Charlie voted yes for this contestant. *1* is therefore added to the "data" element and the operation is furthered to the next node (which is the coordinator in this case). Charlie sent a "GET /get_public_key" request to coordinator and received its public key to be able to encrypt the message.

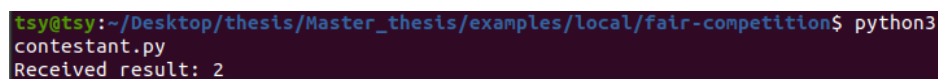


```

47013517725687729397, 65537) from ip: localhost:8082
127.0.0.1 - - [12/Apr/2023 15:28:55] "GET /start_analysis HTTP/1.1" 200 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:28:55] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:28:57] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:28:59] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:29:01] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:29:03] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:29:05] "GET /get_result HTTP/1.1" 404 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:29:07] "GET /get_result HTTP/1.1" 404 -
127.0.0.1 - - [12/Apr/2023 15:29:08] "GET /get_public_key HTTP/1.1" 200 -
Validating request..
Validating certificate
got analysis done
127.0.0.1 - - [12/Apr/2023 15:29:08] "POST /analysis_done HTTP/1.1" 200 -
Received result request
127.0.0.1 - - [12/Apr/2023 15:29:09] "GET /get_result HTTP/1.1" 200 -

```

Figure 5.8: Coordinator's server



```

tsy@tsy:~/Desktop/thesis/Master_thesis/examples/local/fair-competition$ python3
contestant.py
Received result: 2

```

Figure 5.9: Contestant received their vote result

On coordinator's server (figure 5.8) we can see it has received the "POST /analysis_done" request. It validated the request and the certificate, and has subtracted all the scrambles used. When coordinator then receives a "GET /get_result" request from a contestant, it can successfully return the result from the judges votes.

Figure 5.9 shows the contestant receiving the result 2, which means two judges voted yes and one voted no. We don't know which judge voted what (except Charlie since it was purposefully shown). However, we do know that the contestant passed this stage of the competition 2/1 and goes through to the next stage.

More examples, such as calculating correlation using Pearson's R coefficient, can be found in Sneak's documentation in the appendix. It includes cases where input parameters must be given from the client to then be used in the SMC operations.

5.2 Performance

Figure 5.10 - 5.12 below shows how Sneak scales from a network size of 2 up to 128 nodes with milliseconds on the Y axis, and number of requests on the X axis. Figure 5.10 shows sequential requests running. Sequential requests means sending a single analysis requests to the network, waiting for the result before sending the next request. This measure shows the scalability when multiple SMC loops are necessary (for instance with Pearson's correlation). It also shows the baseline for concurrent requests. As expected, sequential should double the time it takes to perform operations. 2 should be twice as fast as 4 and so on. The network is shown to be consistent.

Figure 5.11 and 5.12 shows the scalability when requests are run simultaneously. It no longer shows the time as approximately doubled, as with sequential requests, but rather decreased from the double. With an example from UiT cluster figure 5.12, 64 nodes and 3 concurrent requests, it took about 4.2 seconds. For 6 concurrent requests it took about 5.8 seconds, which is a decrease of about 19% from running sequential requests.

It's worth to note that with these measures from the UiT cluster, nodes are not equal. Some nodes can be quite slow and some quite fast which can differentiate the result. For instance, running 4 concurrent requests should in theory be more similar in terms of time when compared to running 2 concurrent requests. This is because of the idle time between requests on single nodes in a large network is quite big. Therefore handling multiple requests shouldn't increase the time as significantly as it does. However, this is shown to not be the case as nodes have different computer hardware (among other factors shown below with bottleneck). Arguably, the UiT cluster therefore shows a more realistic representation of the scalability, as real nodes usually contain different hardware. Additionally, some nodes might run a lot of other processes simultaneously which can slow down the computing power further.

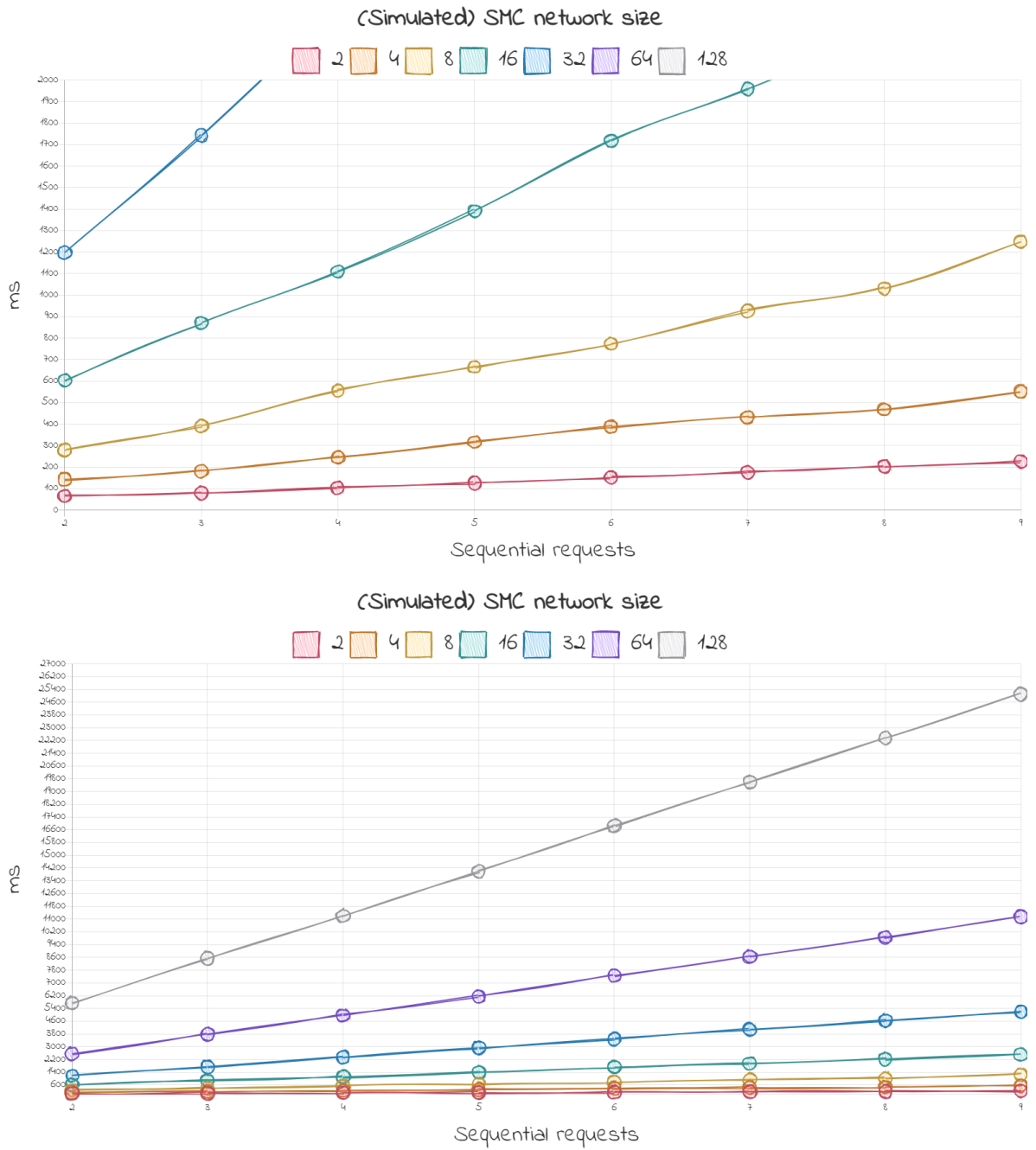


Figure 5.10: Sequential requests running on a simulated Sneak SMC network on a local machine. First image is zoomed in.

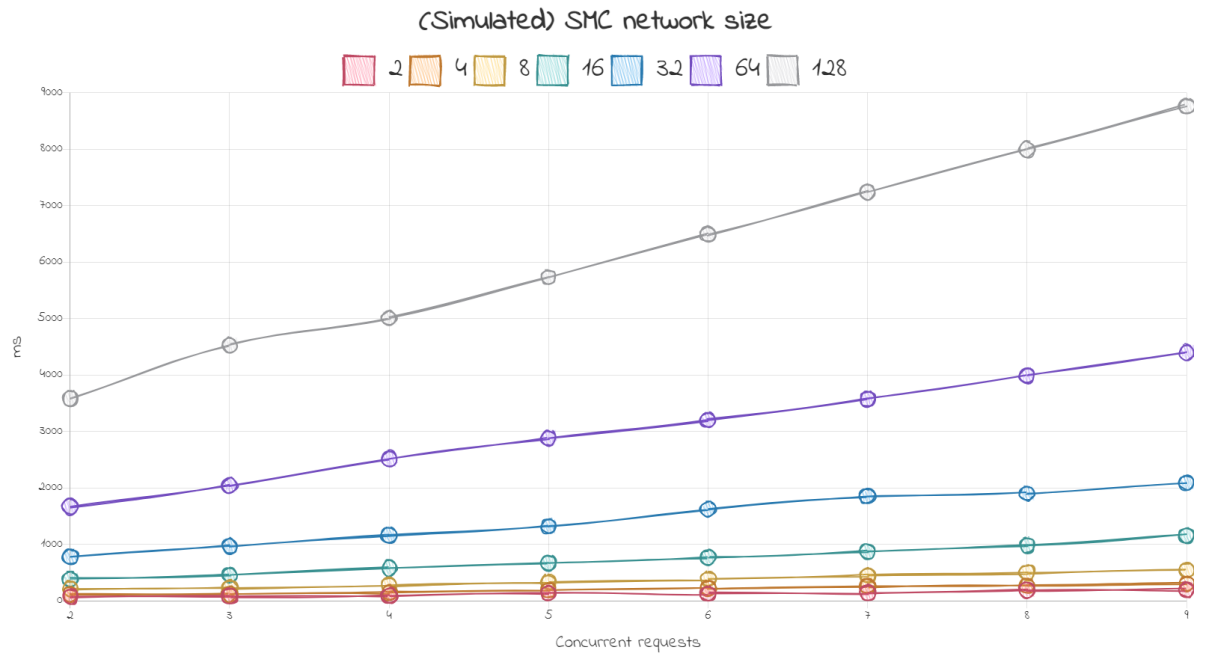


Figure 5.11: Concurrent requests running on a simulated Sneak SMC network on a local machine.

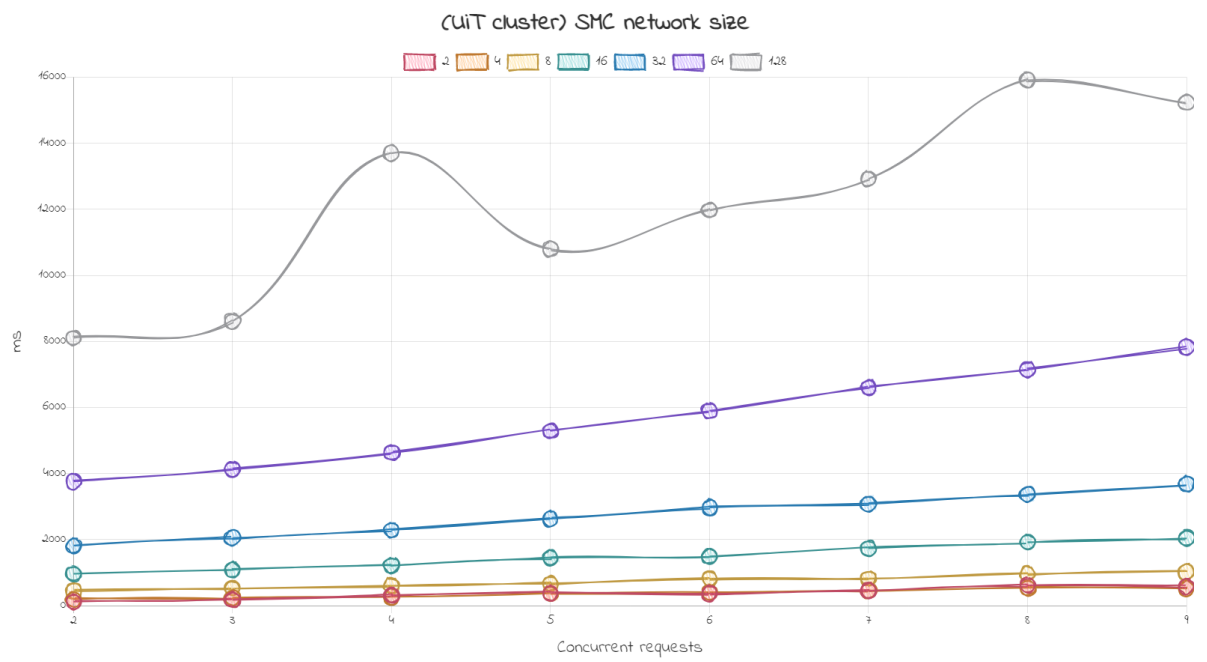


Figure 5.12: Concurrent requests running on a Sneak SMC network on the UiT cluster.

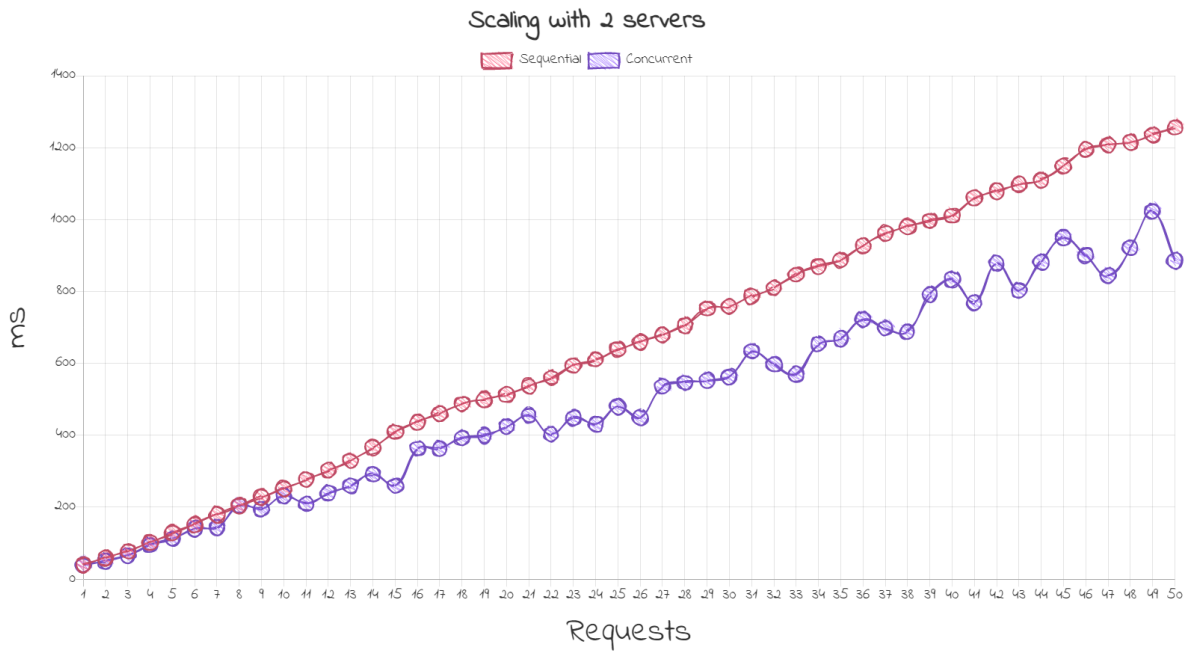


Figure 5.13: Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.

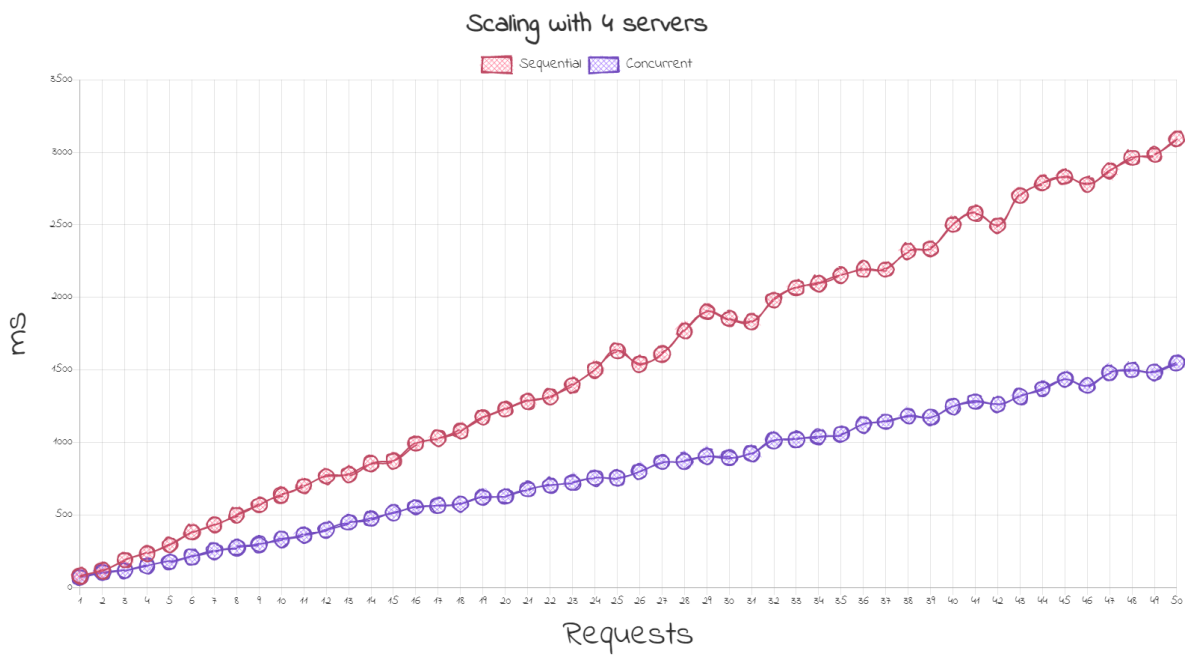


Figure 5.14: Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.

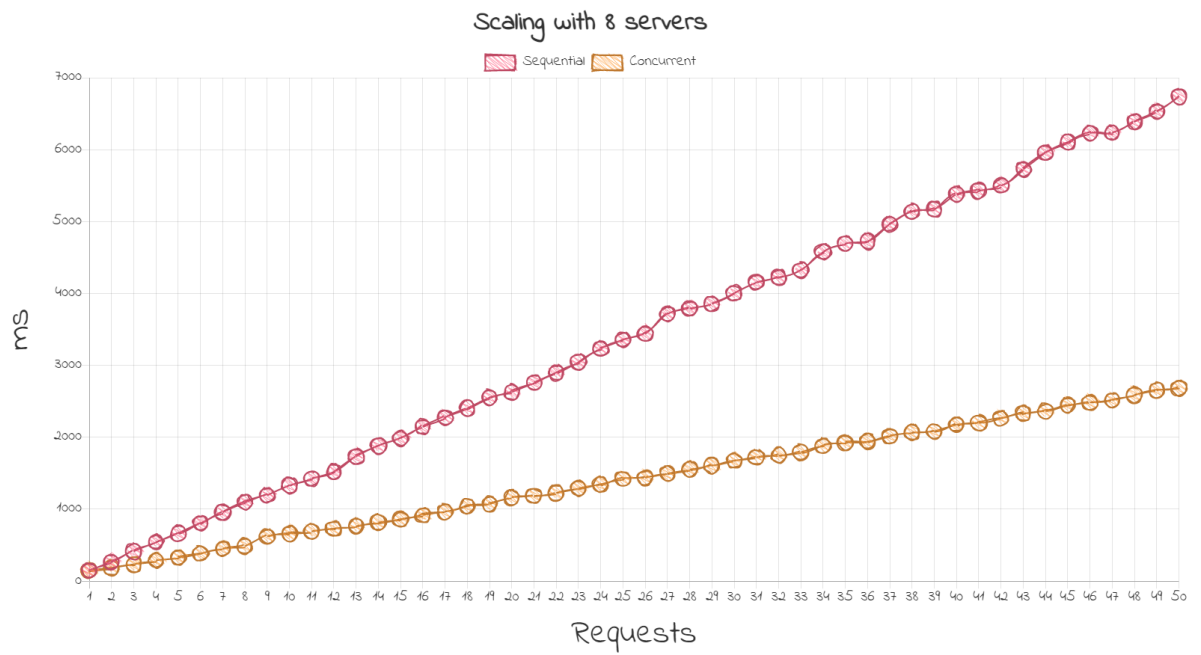


Figure 5.15: Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.

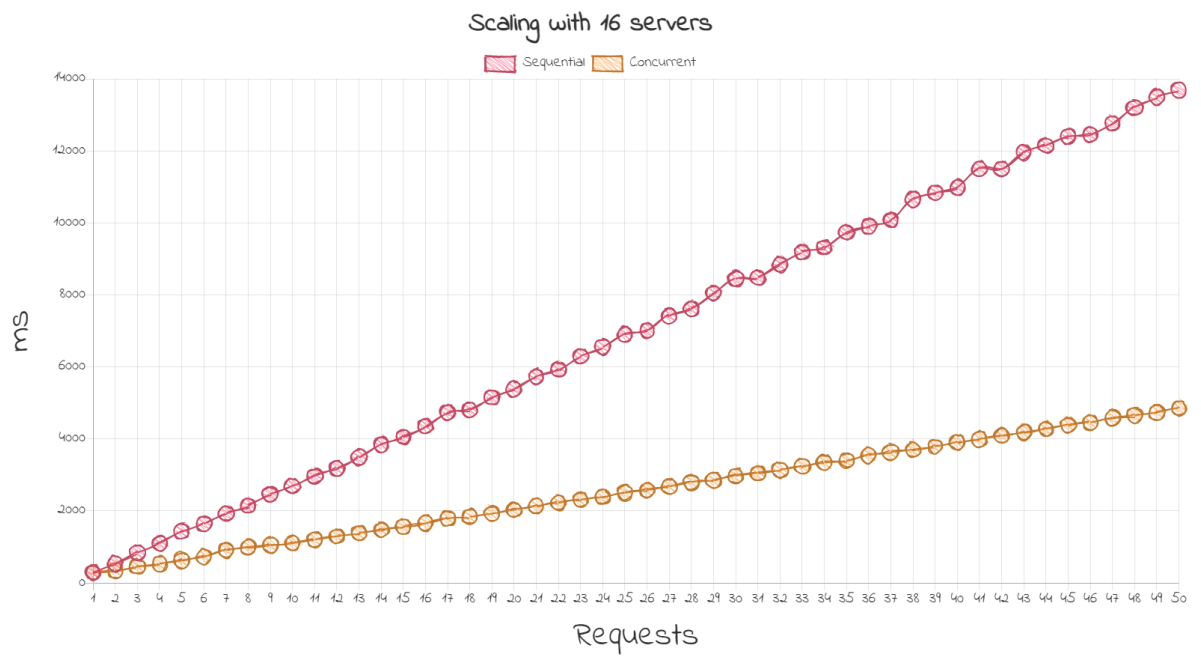


Figure 5.16: Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.

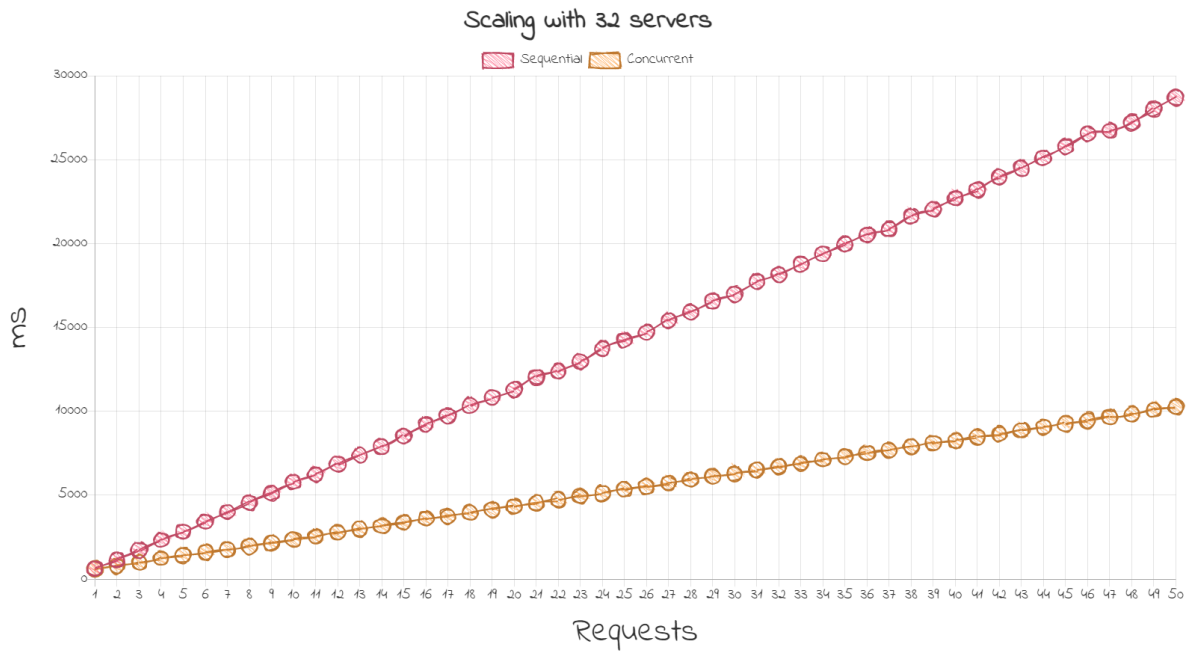


Figure 5.17: Sequential VS Concurrent requests running on a simulated Sneak SMC network on a local machine.

Figures 5.13 to 5.17 show a direct comparison between sequential and concurrent requests. The SMC network size is shown with 2, 4, 8, 16 and 32 servers respectively, with up to 50 requests. From the graphs, the data seems to be approaching a limit where concurrent requests is limited to slightly below half of the speed of sequential requests. With a lower amount of servers (i.e. 2, 3, 4) the concurrent speed is closer to that of sequential requests, which is especially apparent on figure 5.13.

As an attempt to figure out why concurrent requests scaling hit this limit of about 2x(++) performance. The time used for each request on coordinator node was compared to the total time used for the same request. Figure 5.18 shows this comparison. As can be seen, the coordinator takes up a considerable amount of time. Where coordinator time approaches approximately 28% of the total request time (for 128 nodes).

For concurrent requests, this limits the speedup, meaning there is a maximum theoretical increases of $(100-28) 72\%$ per request. However, this assumes perfect conditions where no requests interferes with another, except for on coordinator. However, interference on coordinator does happen, and it's shown to be quite substantial in figure 5.19.

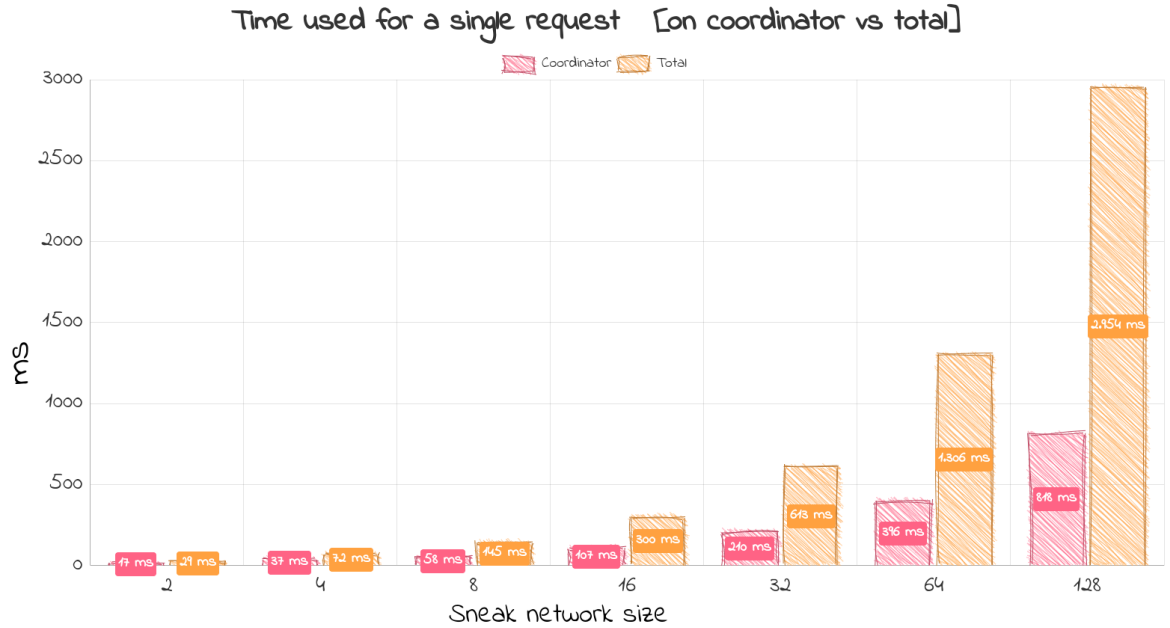


Figure 5.18: Time used on coordinator to initialize a request compared to the total time used on a request. (On local machine)

This figure (5.19) shows how much time X amount of requests uses on the coordinator node alone compared to the total time for all nodes (X is the number of requests on the X axis), with 64 servers. For one request, similar to what figure 5.18 shows, the time is relatively low on coordinator (approximately 28%). Two requests however shows the time almost doubled, and the time increases substantially for each succeeding request. Interestingly, for 50 concurrent requests, the coordinator uses approximately 96% of the total time.

This suggest that the limit seen from figure 5.13 to 5.17 is due to coordinator not being able to initialize SMC operations fast enough. Thus, each concurrent requests is enqueued and blocked on coordinator (a client sending requests will be blocked while waiting for a HTTP response from coordinator). If we assume 28% of the total request time is 350 MS (milliseconds) (based on what we can see from figure 5.19). When coordinator receives two concurrent requests, the second request has to wait at least 350 MS before it can be started. Third request would wait 700 MS and the fourth request would have to wait 1050 MS. This gives the equation $b * (1 + 0.28 * x)$, where b is the base time for a single request and x is the number of concurrent requests minus 1. For 50 requests the equation gives us $1300 * (1 + 0.28 * 50) = 19500$ MS. This corresponds to the data in figure 5.19. The equation makes it possible to calculate how much time a request has to wait before it will be processed in the system. I.e. running 1000 concurrent

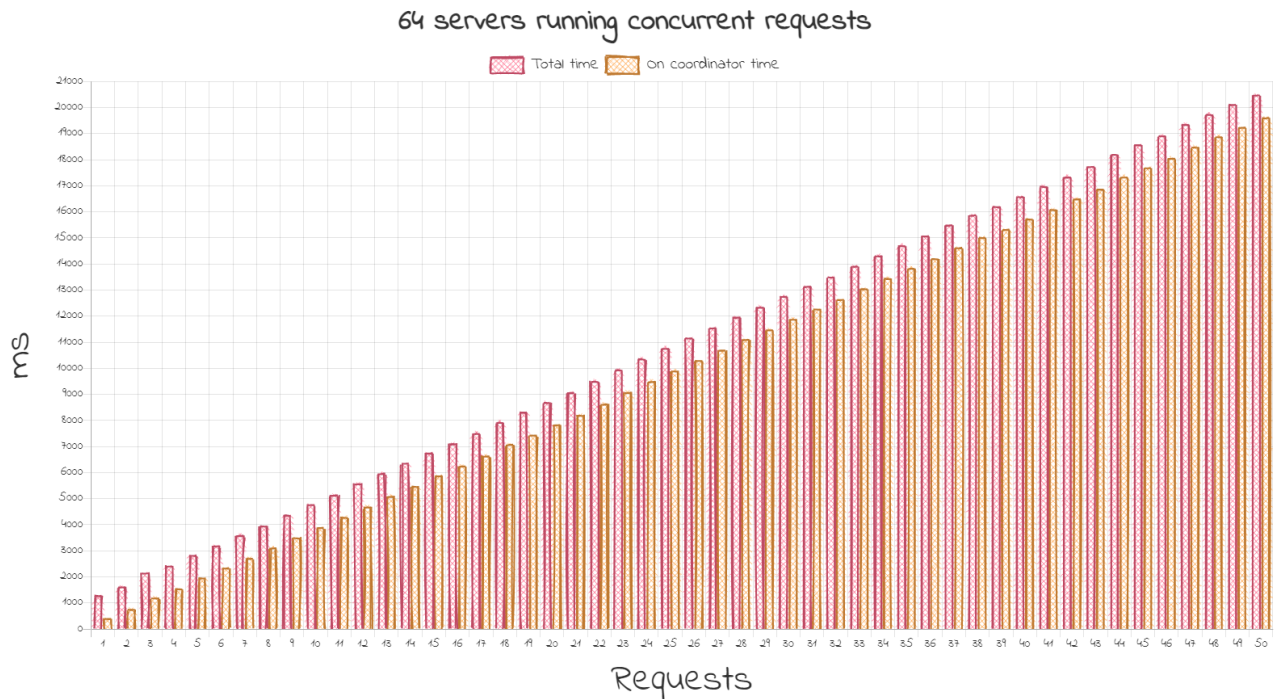


Figure 5.19: Time used on coordinator to initialize X concurrent requests compared to the total time used on X concurrent request (on local machine).

requests, the last request has to wait $1300 * (1 + 0.28 * 1000) = 365300$ MS (365 seconds), due to the bottleneck of the coordinator.

Once the last concurrent request is initialized, most of the other SMC operations is likely done already. The coordinator node seems to be the biggest bottleneck in Sneak. Note, this does not include additional requests the coordinator might receive (such as "get_result"), which would add even more overhead. Additionally, the equation $b * (1 + 0.28 * x)$ is only formed from the data from Figures 5.13 to 5.17.

As an effort to improve aforementioned scalability issue, I tried implementing data compression to reduce the encryption and decryption times. Simultaneously, this reduces data sent from one node to another. Figure 5.20 shows a bar diagram of the result, a single request with and without compression used for different network sizes. Interestingly, the added overhead of performing compression and decompression seems to be consistently higher than without compression altogether. The difference averages approximately 14.6%, and the effort seems to be redundant. The issue lies with coordinator and to improve performance, the work coordinator does should be as little as possible.

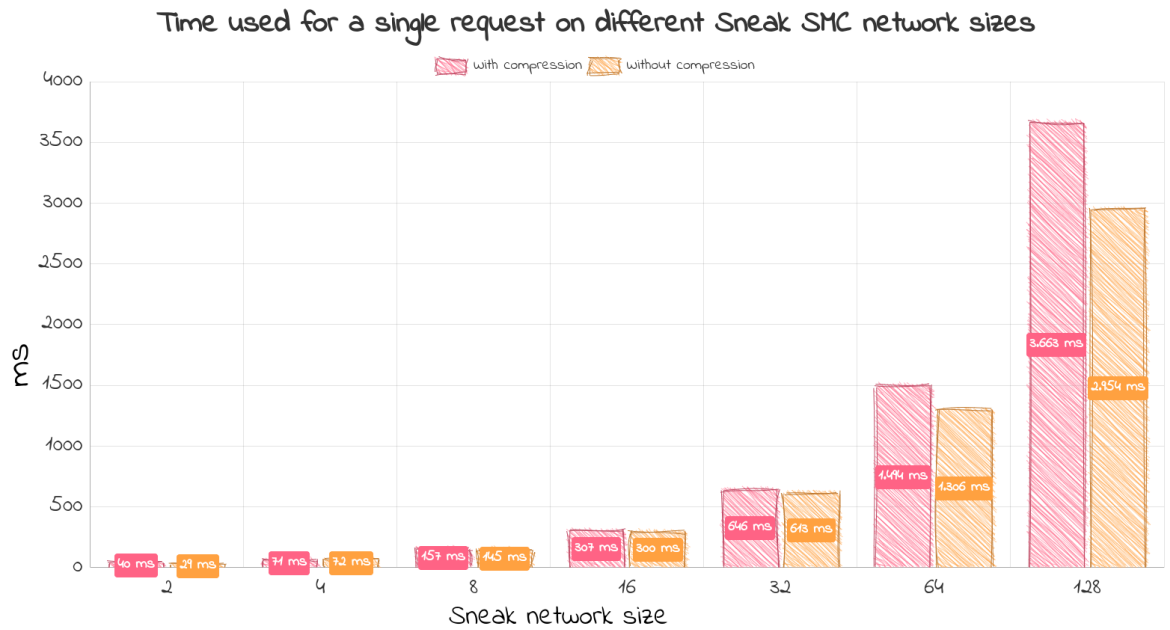


Figure 5.20: Time used for a single request with/without compression. (On local machine)

To sum up the experiments, Figure 5.10 was compared to Figures 5.11 and 5.12 and it showed that running simultaneous requests is faster than sequential requests. It also showed how much faster, which is around twice the speed consistently for higher number of network sizes and number of requests. Figures 5.13 to 5.17 also supports this scalability, but it shows graphs with direct comparison between sequential and concurrent requests. It shows that the speedup for executing concurrent requests hits a limit. A limit due to coordinator not being fast enough to initialize SMC operations and becoming a bottleneck. This causing a chain of delays which blocks and essentially forces sequential execution from that point on. Compressing the data was measured as well with the goal to improve performance but was proved to be less efficient overall.

/6

Discussion

6.1 Experiments

From the experiments taken we can see how Sneak performs on varying sizes. Overall Sneak scales linearly for each request. As can be seen on each figure, running a higher number of SMC network sizes results in higher throughput time. What Figure 5.10 ~ 5.17 effectively shows is a comparison between running requests one by one (sequential) and simultaneously. From these graphs we can see the improvements made, but also some limitations of Sneak, bottlenecks and which parts that can be improved.

For instance, we know that the coordinator is a bottleneck from Figure 5.19, thus the work on coordinator should be relieved as much as possible. When developing Sneak, and before any measurements were taken, a design decision was made to have coordinator create the unique scrambles, encrypt them and send it to each participating node. With a SMC network size of 128 nodes, this effectively means that coordinator has to perform 128 extra computations while also having to encrypt additional data. Originally, I thought that it didn't matter if coordinator was the one who created and encrypted the secret scramble value for each node. After all, it does not matter in terms of information disclosure, network bandwidth and simplicity of the system. However, it has become clear after the measurements from Figure 5.19. that it does in fact matter in terms of efficiency and scalability. Coordinator has to encrypt all of the data, thus has more work than any other node in Sneak. Moving some of the workload from coordinator out to other nodes will therefore improve the system as it

will relieve some of the bottleneck on the coordinator node.

The UiT cluster is a number of distributed nodes that is connected together. Testing Sneak on this cluster showed to be fairly similar to tests on a local machine. It scales linearly as shown in Figure 5.12. There is one noticeable difference however which is more fluctuations in the data. This especially the case when testing with 128 nodes. The theory behind these fluctuations is that nodes have different hardware and therefore operating on different speeds. Interestingly, for size 128 there wasn't enough separate nodes available, such that some nodes would need to handle multiple servers simultaneously in order to hit the quota of 128, whereas some handled only one (as it should be). This would explain the bigger fluctuations that can be seen on the data.

Overall Sneak runs efficiently with higher number of SMC network sizes, where it takes approximately 15 seconds to run 9 requests on a network size of 128 (figure 5.12). It also shows to scale up to (and slightly above) 2x performance from sequential requests.

6.2 Library security

The security of Sneak has been shown to be valid with the use of nodelist, certificates and CAs under the *architecture and design* section. Having strict security measures increases the overall complexity of using Sneak. The complexity could otherwise be simpler but it would not be a feasible solution. It is something that has been investigated but practically difficult to support. One such investigation were if only coordinator needed a valid certificate and all other nodes would not. A situation like this would allow the SMC algorithm to be less computationally heavy. The performance would be improved and subsequently the scalability of the network would be improved. It is however not possible as it leaves too much room for malicious intent, nodes could be impersonated.

Another investigation were if the certificates could be self signed. Thus not having to reach out to a CA and have them sign the certificates. This suffers from the same fault where anyone could create a certificate and sign it. One interesting approach for this issue was to have a CA sign coordinator's certificate and have the coordinator act like the CA for all other nodes in its SMC network. Having nodes with self signed certificates be approved and signed by the coordinator instead. The theory behind this approach was that nodes can trust each other since they already trust the coordinator. Still, the same question would remain, how could coordinator fully trust the nodes if all they have are the self signed certificates? Even if this question could be properly answered,

making coordinator a CA when one already exists seems redundant. Neither this is a feasible solution to improve performance and reduce complexity. It was therefore deemed best to have each node validated by a CA.

The security aspects of sneak has been showed in detail, where potential issues for various conditions has been handled. The faults that can occur was highlighted and presented with figures in the *architecture and design* section. It has been shown that these issues is mitigated by the use of asymmetric encryption, symmetric encryption, IV initialization for symmetric CBC mode, signing and verifying message request and unique scramble for each participating node. Additionally, it has been highlighted why it's so important to trust other fellow nodes in an SMC distributed network.

Otherwise, as long as two or more nodes and the coordinator in the network are clean then the security of Sneak is guaranteed to be reliable and safe. The guarantee of whether a node is clean or not is trusted by a known CA.

Furthermore, to make the library trustworthy, the source code is openly available on github. Usually people don't trust starting a server if they have no idea what it does in the background. Making it open source allows for some level of trust, such that users feel safe using this library. The downside is of course that hackers have access to the same source code and can potentially find a loophole somewhere. Additionally, the source code of Sneak can be inspected after downloading it from pip.

6.3 Library simplicity

We have showed the low complexity of using Sneak, with setting up servers, shutting them down and running SMC operations. The client submodule can start an operation and retrieve the result directly in the terminal window. Otherwise, the client can be imported in a python script and do the aforementioned in two lines of code.

We have showed how servers are set up initially and the complexity of what has been done in the background which users can't see. The coordinator server can be imported and run with 1 line of code, otherwise it can also be run directly from the terminal window, similarly to the client. Normal servers must import the Sneak module to a custom python script in order to run, but does so to hide a vast amount of needed complexity in the background of SMC operations. Servers can still be run with only a few lines of code depending on the size of the analysis function.

We have showed an example of using Sneak to solve the fair competition problem. This example shows in detail the usage of Sneak and the low complexity it offers for handling otherwise complex operations. Setting it all up took merely a few lines of code that could handle unlimited amount of unique vote rounds (SMC operations), and it could then guarantee the complete anonymity of a judge's vote.

We have showed how Sneak can be used as a general medium to run SMC operations with the use of custom analysis functions. To have the analysis function user defined means that users can return results of more complex types (if they need), such as a matrix of different values, rather than a simple integer. The desired simplicity will still be achieved, where the added complexity of scrambling complex data types is hidden from users.

Finally, we have showed the simplicity of using the cryptography module when communicating with other servers. Users doesn't have to know that it exist, as such, the cryptography module is abstracted away completely from users. The complexity of ensuring valid communication requests is otherwise quite substantial, which includes encrypting, signing and verifying certificates in addition to other metadata. This is between each request that is sent and received. It is all handled internally.

6.4 Library scalability

The last goal of Sneak was to keep the level of scalability within a practical frame of use. However, the scalability by default has an issue on single cycle operations and that is that it can only execute the analysis of one node at a time (single cycle execution). Meaning that each node has to wait for the previous node to finish before the next will receive an analysis request. Slow nodes will therefore slow down the process for all other nodes. The overall execution time of single operations will increases quite drastically for each node participating in the SMC network (see figure 5.10). This is the case for each operation running, and it must be this way to maintain the round-robin structure of nodes and to ensure optimal levels of security.

To battle some of these constraints the servers in Sneak utilize a threading variant of *BaseHTTPRequestHandler* [20, 21] to support concurrent operations. Which means that a SMC network can run multiple operations simultaneously if the system receives two or more different "start_analysis" requests from clients. Since single cycle execution hurts the performance most (at least for larger networks), Sneak improves scalability by reducing the amount of loop operations needed to be run, thereby reducing the single cycle execution to as few

as possible. Where one loop is the most optimal to keep the minimal desired overhead. This factor is heavily influenced however by the users need. For instance to calculate Pearson's coefficient, a minimum of two SMC operations is needed regardless. The first must be done to retrieve the mean value of some data. And the second iteration is then able to calculate the data coherency with Pearson's R coefficient, where the mean input was given from the previous iteration.

As can be seen in the *experiments* section, reducing the loops from two to one, or more drastically from four to one, increases the overall efficiency by substantial amounts for single cycle execution operations. For larger networks, reducing loop iterations is the best way to improve the scalability aspects. This design has shown an optimal way of running SMC operations where all analysis is done efficiently within one loop iteration. However, there are specific cases where multiple loops are unavoidable, such as with Pearson's coefficient.

Other aspects of scalability is that the workload is naturally split among each node. Only the node itself can perform the analysis on its own data, as such the work will be balanced. Therefore running multiple concurrent SMC operations will not slow down the execution time excessively. The scalability of Sneak is shown to be improved when dealing with simultaneous requests. This can however be influenced by users implementation of the analysis function. As an example with the fair competition problem shown in the *experiments* section, this is a very special case where it can only handle one operation at a time since it waits for user's input. Special cases like this are rare but can occur.

6.5 Computational issues with Sneak

As much as a general SMC solution is preferred, it is very hard to achieve flawlessly. The solution mentioned in this paper can't perform certain special SMC operations optimally and effectively. The millionaires problem is one of these instances where it would be tedious and inefficient to perform the calculations needed. Not to mention that it will partially disclose information to other nodes. To implement the millionaires problem with Sneak, the two nodes must be asked, e.g. do you have 100 millions? They must supply a yes-no answer, and if coordinator receives two no's then it must ask, do you have 50 millions? This is done until there is no more money left to differentiate the two. If now coordinator receives a 1-1 result then it can broadcast the result to the two nodes. Both of the nodes will know that one has above 50 million and one has below 50 million. The range of the disclosed data d will be $0 \leq d < 50$ and $50 \leq d < 100$ (millions).

It might be an acceptable range, however, consider instead of their wealth being differentiated between 50 million to be differentiated between 10 000. This might not be an acceptable range. It can also be very costly to perform as the number of SMC operations needed to calculate the result has worst case scenario $O(\log_2 M)$. This is expressed with big O notation where M is the starting number used to ask participants. With M=1-billion, by continuously splitting the half (log base 2) it could potentially result in 30 SMC operations. As you can see, the work and the disclosed information can vary immensely. Therefore it is not an optimal strategy for this type of problem with the current implementation of Sneak.

What Sneak is optimized and works well for however are scenarios where the intention is to keep everything anonymous, either by keeping the result hidden or having individual values hidden by scrambling. These types of operations work well with relationship between data. It can be used within healthcare analysis, joint database analysis and voting structures to name a few.

6.6 Preventing man in the middle attack

The most probable attack to a system using public key structure such as Sneak is man in the middle attack (MitM). If Alice wants to connect to a coordinator (Bob), she sends her public key to Bob. However, the message is caught by an attacker (Eve) who swaps Alice's public key to her own and re-transmits the message to Bob. On initialization neither party knows each others public key such that the first message must be unencrypted, opening up the possibility for MitM. Bob has no way of knowing that he received Eve's public key and not Alice's. Eve can now continue to pick up messages that is transmitted between Alice and Bob, decrypt it to be able to read its contents, re-encrypt the message and then re-transmit them to the other party. This type of attack is prevented by the use of a certificate along with a trusted CA. Since the message is signed by a trusted CA, when Alice receives a message from Eve (who she thinks is Bob), she can validate that the certificate she received does not correspond to the sender who sent it, thus catch the MitM attack. Alice prevents sharing any private data by ignoring the message completely and drop the communication session with Eve.

Preventing such attacks is important in Sneak to maintain valid and secure communication sessions. Even if eve managed to insert herself between two or more nodes communicating, she will still not understand the sensitive data as it has been scrambled. The only way to interpret it is if she has the secret scramble value that was sent from the coordinator.

6.7 Onion encryption vs partial encryption

When coordinator node sets up the SMC operation it has a choice on how it proceeds with delivering information securely. With onion encryption (or layer encryption) there will be layers on top of layers of encryption. This means that node 1 can decrypt the message and only be able to read its own message. The rest is still encrypted with node 2's public key (another layer). Then node 3 depending on how many nodes there are in the system. To make this system work, the order of who decrypts when matters as the coordinator node sets a specific sequence when encrypting the data.

Partial encryption don't need this ordering as the encryption is separated. It requires less encryption power and will therefore also run faster because of it. The only issue that can make onion encryption preferred is that other nodes won't know how many nodes are left to decrypt messages. In partial encryption there is a list of encrypted data which eventually is popped from the lists as nodes reads the information they need and furthers the list to the next node. This can be seen on figure 4.1 in the *architecture and design* section. Information which is not needed for nodes could therefore be learned by examining the list. This would not be the case with layered encryption.

6.8 Future work

6.8.1 Automacy

The broadcasting of the whitelist allows for an added feature where if the coordinator node crashes, then another node can take its place. To be able to automatically choose a new coordinator, all nodes must agree to the change, including the client (connected to coordinator). Generally, small distributed networks does not need this added overhead as nodes will rarely crash or have latency issues among each other. In the case that it does, it will be easy to reboot. Bigger systems with millions of nodes might always assume some node has crashed or doesn't respond due to connectivity issues. Here, automacy could be essential. An issue is when a client tries to communicate with a crashed coordinator. Normally this won't work, however, the crashed node should eventually be rebooted again such that the client can continue to communicate with it without any interference. If this happens then it's possible to update the coordinator address for the client, such that it can re-correct itself. For instance, the crashed coordinator node sends the new coordinator address back to the client. It can take some time before a node is able to be rebooted and the election of a new coordinator node might introduce complex consensus algorithms. This is therefore mentioned here in future work, as an idea of how to progress and au-

tomate the SMC operations and prevent faults that can occur. It's even possible to introduce another mode to this SMC design, specifically for larger systems which require extra work, and thus causing more complexity when starting up. Segregating it from the current design could keep the desired simplicity for smaller systems and having an option for larger ones.

6.8.2 Databases

The whitelist doesn't need to be stored in a traditional database like MySQL. It could just be a simple file that acts like a database (as with the current design). However, this depends on the size of the distributed system. If the system has 10 nodes, then a simple file is more than good enough. If the system has 1 million nodes, then the file overhead might be too big and a database will improve disk i/o performance. The database could additionally store operation ID's and its results, as well as executable code for the analysis functions. However, this has not been explored during the work of this thesis, but could be relevant for future work when improving performance for large scaling networks.

6.8.3 Executable code

Other future work is having coordinator send the executable code to perform the analysis on data. This would eliminate the need to define specific analysis functions for each node. Instead they will receive the work by the coordinator. The upside for this scheme is that a server could be run directly from the terminal (for instance as *sneak runserver -port 8080*). It would allow for dynamically altering the analysis functions without having to restart any server. To achieve this sufficiently, the execution code could be supplied by a client to the coordinator, and then by coordinator broadcasted to the entire network of nodes. Additionally, this allows a client to specifically query the information it needs directly. It can provide the correct input and define the output data structure it wants. The only restriction this imposes is that the analysis data must be uniformly stored on each node. Otherwise, the executable code must handle the analysis differently for each node, which can make the analysis function overly complex and difficult to implement. Ultimately, if nodes have the same analysis function, then sending executable code allows for higher degree of simplicity and usability.



Conclusion

SMC is a technique for performing computations on data without revealing the data itself. This technique is particularly useful when dealing with sensitive data or when parties do not trust each other with their data. The Sneak library is a practical implementation of SMC that has been shown to provide secure handling of SMC operations over a distributed network of nodes. It is optimized for handling data where anonymity is important, such as in healthcare analysis, joint database analysis, and voting structures. The way it achieves anonymity is by scrambling individual values, and as more nodes add their results to a sum, making it increasingly difficult to read. Sneak's architecture and implementation have been presented, which includes a round-robin structure that allows the coordinator to initialize random and unique operation orders and unique scramble values given to each node. The coordinator also communicates with clients, allowing them to start new operations, retrieve results, and shut down the network.

Furthermore, the design details and issues of Sneak have been highlighted, and the mitigations it implemented to prevent faults and insecure operations were demonstrated. Examples were provided to show how Sneak can perform secure analysis on sensitive data involving a network of distributed nodes. With Sneak, sensitive data can be analysed securely without compromising the privacy of the participants. Sneak is a powerful tool for performing secure computations, and it can be easily integrated into existing systems to ensure privacy and security. Through detailed examples, Sneak's capabilities have been demonstrated, and its faults have been mitigated to improve its security.

Bibliography

- [1] Andrew C. Yao. Protocols for secure computations. [Available at <https://ieeexplore.ieee.org/document/4568388>]. *University of California Berkeley, California*.
- [2] Merete Saus Anders Andersen. Privacy preserving distributed computation of community health data. [Available at <https://www.sciencedirect.com/science/article/pii/S1877050917317295>]. *University of Tromsø, Norway*.
- [3] Wenliang Du and Zhijun Zhan. A Practical Approach to Solve Secure Multi-party Computation Problems. [Available at <https://dl.acm.org/doi/10.1145/844102.844125>]. *Syracuse University*.
- [4] Lawrence Williams. Round robin scheduling. URL <https://www.guru99.com/round-robin-scheduling-example.html>.
- [5] Adi Shamir. How to share a secret. [Available at <https://dl.acm.org/doi/10.1145/359168.359176>]. *ACM digital library*.
- [6] Tiina Turban. A Secure Multi-Party Computation Protocol Suite Inspired by Shamir's Secret Sharing Scheme. [Available at <https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/262970>]. *NTNU*.
- [7] Zheng-an Yao Chunming Tang. A New (t,n) -Threshold Secret Sharing Scheme. [Available at <https://ieeexplore.ieee.org/abstract/document/4737091>]. *IEEE*.
- [8] Kazuhide Fukushima Toshiaki Tanaka Jun Kurihara, Shinsaku Kiyomoto. A New (k, n) -Threshold Secret Sharing Scheme and Its Extension. [Available at https://www.researchgate.net/publication/220905280_a_new_k_n_threshold_secret_sharing_scheme_and_its_extension]. *Research gate*.
- [9] Anders Andersen. SNOOP: privacy preserving middle-

- ware for secure multi-party computations. [Available at <https://dl.acm.org/doi/10.1145/2677017.2677025>]. *University of Tromsø, Norway*.
- [10] Yehuda Lindell. Secure Multiparty Computation (MPC). [Available at <https://dl.acm.org/doi/10.1145/3387108>]. *ACM digital library*.
- [11] Inpher. What is secure multiparty computation. URL <https://inpher.io/technology/what-is-secure-multiparty-computation/>.
- [12] Shafi Goldwasser. Multi-party computations: past and present. [Available at <https://dl.acm.org/doi/10.1145/259380.259405>]. *Conference: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*.
- [13] Daniele Francioli. General data protection regulation (gdpr) and location data. URL <https://joinup.ec.europa.eu/collection/elise-european-location-interoperability-solutions-e-government/news/gdpr-and-location-data>.
- [14] Stacey Grey. A closer look at location data: privacy and pandemics. URL <https://fpf.org/blog/a-closer-look-at-location-data-privacy-and-pandemics/#:~:text=Precise%20location%20data%20is%20legally,requirement%20of%20affirmative%20express%20consent>.
- [15] Google. Google privacy and terms. URL <https://policies.google.com/technologies/location-data>.
- [16] Dennis Byrne. *Full stack python security*. Manning publications, 2021. ISBN 9781617298820.
- [17] Sharon Shea Alexander S. Gillis. X.509 certificate. URL <https://www.techtarget.com/searchsecurity/definition/X509>.
- [18] Toby Gaff. Certificate chain of trust. URL <https://www.keyfactor.com/blog/certificate-chain-of-trust/>.
- [19] computer security resource center Nist. Transmission control protocol (tcp). URL https://csrc.nist.gov/glossary/term/transmission_control_protocol.
- [20] Python documentation. http.server — http servers, . URL <https://docs.python.org/3/library/http.server.html>.
- [21] Python documentation. Asynchronous mixins, . URL <https://docs.python.org/3/library/socketserver.html#asynchronous-mixins>.

- [22] Diego Barba. Python concurrency — threading and the gil. URL <https://towardsdatascience.com/python-concurrency-threading-and-the-gil-db940596e325>.



Appendix

Sneak documentation

May 2023

Contents

1	Cryptography submodule	2
1.1	Public methods	2
1.1.1	Cryption.key2string(key)	2
1.1.2	Cryption.string2key(strkey)	2
1.1.3	Cryption.get_keysize()	2
1.1.4	Cryption.get_public_key()	3
1.1.5	Cryption.encrypt(to_user_pkey, str_msg)	3
1.1.6	Cryption.decrypt(cipher)	3
1.2	Private methods	4
1.2.1	Cryption.__init__(self)	4
1.2.2	Cryption.generate_key(key_size)	4
1.2.3	Cryption.Pad_message(msg)	4
1.2.4	Cryption.Remove_pad(msg)	5
1.2.5	Cryption.sign(msg)	5
1.2.6	Cryption.verify(key, msg, signature)	5
2	Client submodule	7
2.1	Functions	7
2.1.1	send_client_request(type, url, body, receiver, timeout)	7
2.2	Client class methods	7
2.2.1	Client.__init__(self)	7
2.2.2	Client.request_analysis(coordinator_address, data)	7
2.2.3	Client.get_result(id, tries, ms)	8
2.2.4	Client.shutdown(node)	8
3	Server submodule	10
3.0.1	Server.run_server(coordinator, analysis_function, nodefile, ip, port, timeout, request_timeout, min_nodes)	10
4	Examples	11
4.1	Example 1	11
4.2	Example 2	12

1 Cryptography submodule

This module consist of a single class called **cryption** and can also be run as a main script to test its correctness. Meaning only running "python3 crypt.py". The functions is divided into public and private. All private functions is internally handled and is not necessary to know about when importing this module.

1.1 Public methods

1.1.1 `Cryption.key2string(key)`

Converts a key object to string format in order to send it easily over a network and reconstruct it on the receiving side.

- Parameters
 - @ **key**: The key to be converted to string.
- Returns
 - ← The CryptKey as string

Note: key2string is a static class method.

1.1.2 `Cryption.string2key(strkey)`

Converts string to a key object defined within the RSA library. The strkey must be created from the `@Cryption.key2string` method.

- Parameters
 - @ **strkey**: The string key to be converted to CryptKey object.
- Returns
 - ← CryptKey object.

Note: string2key is a static class method.

1.1.3 `Cryption.get_keysize()`

Returns the key size used in bytes.

- Parameters
- Returns

← Integer.

1.1.4 `Cryption.get_public_key()`

Returns the public key object on this instance.

- Parameters
- Returns
 - ← CryptKey object.

1.1.5 `Cryption.encrypt(to_user_pkey, str_msg)`

Encrypts given message and adds meta data. Generates a symmetric key used to obscure the data along with a random IV. Encrypts the symmetric key to the given public key and signs the encrypted message. The return value is a sequence of bytes which includes its own public key, such that the receiver is able to validate the message by comparing the signature hash. Returns public-key-size(4 bytes) + IV + public-key + signature + key + encrypted-msg, as a byte stream.

- Parameters
 - @ `to_user_pkey`: The public key object.
 - @ `str_msg`: The message to be encrypted to the public key `to_user_pkey`.
- Returns
 - ← Byte stream sequence.

Note: Metadata is added to the returned byte stream. It can only be decrypted by the `@Cryption.decrypt` method.

1.1.6 `Cryption.decrypt(cipher)`

Decrypts the encrypted message from encrypt function. First reads all values from the byte stream. Then decrypts the secret symmetric key and uses it to decrypt the message. The senders public key is converted from string to a key object and used to verify the message (by checking hash). Returns the decrypted message.

- Parameters

@ **cipher**: The encrypted message cipher from the **@encrypt** method.

- Returns
← The decrypted string.

Note: This method can only decrypt encrypted messages from the **@Cryption.encrypt** method.

1.2 Private methods

1.2.1 **Cryption.__init__(self)**

Initializes self by creating the public and private RSA keys used for encryption/decryption of data.

- Parameters
- Returns
← self.

1.2.2 **Cryption.generate_key(key_size)**

Generates an unique random symmetric key (used for each message encrypted).

- Parameters
@ **key_size**: The key size in bytes.
- Returns
← Key.

1.2.3 **Cryption.Pad_message(msg)**

Pads a message to a byte multiple of size 16. Since symmetric encryption obscures blocks of 16 bytes per iteration. Note this block size can be changed but must be similar to the key size used. This library uses key size of 16. The padded message is the null-terminator sign '\0' to ensure that it won't ever be used in the actual message content and cause wrong message to be received.

- Parameters

@ **msg**: The message to be padded.

- Returns
← The new padded message.

Note: This is a static method.

1.2.4 **Cryption.Remove_pad(msg)**

Removes the pad used in the msg.

- Parameters
@ **msg**: The padded message.
- Returns
← The message without padding.

Note: This is a static method.

1.2.5 **Cryption.sign(msg)**

Sign by hashing the message and encrypting it with the secret key. Anyone with the signers public key can decrypt and compare the hashed value, to verify that the message is from the correct sender and hasn't been altered during transmission. Hashes is performed using the SHA-256 algorithm.

- Parameters
@ **msg**: The message to sign.
- Returns
← The signed message.

1.2.6 **Cryption.verify(key, msg, signature)**

Verifies the signature with the message and public key given. Returns true or false.

- Parameters
@ **msg**: The message to sign.

- Returns

- ← True if @msg was signed by given public key.

- ← False if @msg was not signed by given public key.

2 Client submodule

This module consist of one function `@send_client_request` and one class `@Client()`.

2.1 Functions

2.1.1 `send_client_request(type, url, body, receiver, timeout)`

Sends a request to another node/server. Returns HTTP response code and a message reply as a tuple.

- Parameters

- @ **type**: The http request type ("GET", "POST", etc..). Default type is "GET".

- @ **url**: The url that should correspond to this request.

- @ **body**: The message body to send (string).

- @ **receiver**: The server address which will receive this request, in "ip:port" format.

- @ **timeout**: The request timeout in seconds. Should be applied in case server can't be reached.

- Returns

- ← (Status-code, Message) as tuple object

- ← (None, None) as tuple object on error.

Note: Blocks until response is received.

2.2 Client class methods

2.2.1 `Client.__init__(self)`

Initializes a sneak cryption object.

- Parameters

- Returns

- ← The Client object.

2.2.2 `Client.request_analysis(coordinator_address, data)`

Sends a request to start a Sneak SMC analysis operation to the given coordinator node. If the coordinator address given is not coordinator then the request will be

simply dropped by the SMC network. Returns (response-code, response-msg) tuple from coordinator. Raises exception on error.

- Parameters

- @ **coordinator_address**: The address for the leader of the Sneak SMC network, in "IP:port" format.

- @ **data**: Additional data supplied to coordinator, such as input arguments for special SMC operations.

- Returns

- ← (200, <operation-id>) tuple received from coordinator if successful.

Note: Raises exception on error.

2.2.3 Client.get_result(id, tries, ms)

Sends HTTP request to coordinator node for the result of a requested analysis operation. Returns the result (if ready) as (response-code, response-msg) tuple, otherwise if not received, returns from last message sent. Raises exception on error. Returns immediately if given operation_id is not valid.

- Parameters

- @ **id**: The operation ID received from @request_analysis method.

- @ **tries**: Specifies the max number of get-requests sent until giving up.

- @ **ms**: Specifies the sleep time in ms between each request try.

- Returns

- ← (200, <result>) tuple if successful.

- ← (<error-code>, <error-message>) tuple if fail.

Note: Raises exception on error.

2.2.4 Client.shutdown(node)

Shutdown given node (or entire SMC network if coordinator node is given), in "IP:port" format. If node=None then tries to shut down coordinator from last analysis request sent.

- Parameters

@ **node**: The node address in "IP:port" format to shut down. Node=None will shutdown coordinator from previous @request.analysis sent.

- Returns

← (200, "ok") tuple if successful.

← (<error-code>, <error-message>) tuple if fail.

Note: Raises exception on error.

3 Server submodule

3.0.1 `Server.run_server(coordinator, analysis_function, nodefile, ip, port, timeout, request_timeout, min_nodes)`

Shutdown given node (or entire SMC network if coordinator node is given), in "IP:port" format. If node=None then tries to shut down coordinator from last analysis request sent.

- Parameters

- @ **coordinator**: The coordinator address.

- @ **analysis_function**: The analysis function.

- @ **nodefile**: The nodefile, list of node "IP:port" allowed in the system.

- @ **ip**: The server IP address to use, "localhost" for local ip, or None to automatically find the public IP address for this machine.

- @ **port**: Port to use, or None to automatically find a port.

- @ **timeout**: The timeout until server shuts down automatically, or None for the default 2 hours.

- @ **request_timeout**: The Timeout for each request sent in order to prevent any type of hang ups on error/unavailable nodes.

- @ **min_nodes**: The minimum amount of nodes needed for a valid SMC operation.

- Returns

- ← Nothing.

Note: Raises exception on error.

Note: Only coordinator needs nodefile and min_nodes. Coordinator does not need to define an analysis function.

4 Examples

Two examples will be shown, the first is a simple example and the second will more complex taking input argument from a client.

4.1 Example 1

This example will show how to compute a simple sum of private data with three parties involved. First, all three servers must be set up and run. Listing-1 shows how this is done.

```
1 import sneaksmc.server as sneak
2
3 def get_sum():
4     return 100
5
6 # The analysis function
7 def sum_analysis(prev_sum, input_argument):
8     my_sum = get_sum()
9     return prev_sum+my_sum
10
11 if __name__ == "__main__":
12     # Coordinator address
13     caddr = "localhost:8089"
14
15     sneak.run_server(analysis_function=sum_analysis, coordinator=
16                     caddr, ip="localhost")
```

Listing 1: Setting up servers

Next, the coordinator server must be run. Listing-2 similarly shows the code for this.

```
1 import sneaksmc.server as sneak
2
3 if __name__ == "__main__":
4     nodes = "nodefile.txt"
5     sneak.run_server(nodefile=nodes, ip="localhost", port=8089,
6                     min_nodes=3)
```

Listing 2: Setting up coordinator

Coordinator must supply a nodefile of all nodes accepted in the system. A nodefile should look like Listing-3 for this example.

```
1 localhost:8080
2 localhost:8081
3 localhost:8082
```

Listing 3: Example of a nodefile

Now Sneak has been initialized and is ready to receive operation requests from clients. Listing-4 shows the client code to send requests and receive results.

```
1 from sneaksmc.client import Client
2
3 if __name__ == "__main__":
4     coordinator = "localhost:8089"
5     c = Client()
6     code, id = c.request_analysis(coordinator)
7     code, res = c.get_result(id)
8
9     # Should print 300
10    print("Sum result: %s" % res)
```

Listing 4: Client starting a SMC operation and receiving the result

The client will receive an operation ID from the coordinator after sending a *request_analysis* request. A coordinator might have multiple simultaneous requests running. Therefore, this ID must be given to correctly identify which operation to receive the result for.

When a client is done and the SMC network will not be used anymore, client can send a shutdown request to coordinator, which subsequently will shutdown all other nodes as well. Example is shown in Listing-5.

```
1 from sneaksmc.client import Client
2
3 if __name__ == "__main__":
4     coordinator = "localhost:8089"
5     c = Client()
6     code, id = c.request_analysis(coordinator)
7     code, res = c.get_result(id)
8
9     # Should print 300
10    print("Sum result: %s" % res)
11
12    # Clean up
13    c.shutdown(coordinator)
```

Listing 5: Client shuts down the SMC network after running an operation

4.2 Example 2

This example will show how to compute correlation between private data using Pearson's R coefficient with three parties involved. Pearson's R coefficient is special as it requires the average before it is able to calculate the result. This means that we must run two loops in our SMC network (2 operations). The first must retrieve the average. This example will continue from Example 1 where we divide the sum data by 3 to gain the average.

Setting up all the servers will be exactly identical, however, we need to adjust the analysis function. The new function is shown in Listing-6.

```

1 import sneaksmc.server as sneak
2
3 def get_sum():
4     return 100
5
6 def calculate_pearson(average):
7     # Perform the calculations
8     ...
9     return data
10
11 # The analysis function
12 def sum_analysis(prev_sum, input_argument):
13     if input_argument is not None:
14         return prev_sum + calculate_pearson(input_argument)
15     else:
16         my_sum = get_sum()
17         return prev_sum+my_sum
18
19 if __name__ == "__main__":
20     # Coordinator address
21     caddr = "localhost:8089"
22
23     sneak.run_server(analysis_function=sum_analysis, coordinator=
24                     caddr, ip="localhost")

```

Listing 6: Setting up servers for handling two SMC operations

Now each server is able to handle both SMC operations, one for calculating the average and the other for calculating Pearson's R coefficient.

The client must also give the input variable for the second SMC operation. An example is shown in Listing-7.

```

1 from sneaksmc.client import Client
2
3 if __name__ == "__main__":
4     coordinator = "localhost:8089"
5     c = Client()
6
7     # Retrieving the average
8     code, id = c.request_analysis(coordinator)
9     code, average = c.get_result(id)
10
11    # Retrieving the data correlation
12    code, id = c.request_analysis(coordinator, average)
13    code, correlation = c.get_result(id)
14
15    print("Correlation between data: %s" % correlation)
16
17    # Clean up
18    c.shutdown(coordinator)

```

Listing 7: Client running two operations

