Secure Multiparty Computation: Protocol Comparison

Gareth Michael Foster
B.A.(Mod.) Computer Science
Final Year Project  April 2013
Supervisor: Prof. Stephen Barrett

School of Computer Science and Statistics
O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

____________________
Gareth Foster, 24th April 2013
Permission to lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_____________________
Gareth Foster, 24th April 2013
Acknowledgements

Firstly I’d like to thank my Supervisor, Prof. Stephen Barrett, for initially suggesting this project to me and providing useful advice and encouragement throughout the project.

I’d also like to thank my girlfriend, Eva Leonard, who has supported me throughout this project and my final year in college, and has provided me with great company and moral support whenever it was necessary.

More thanks are needed for my two good friends and classmates, Kieran Manning and David Ruane, both of whom played a big part in keeping me sane throughout this last year.

Finally I’d like to thank all of my friends and family for the support and encouragement they’ve given that was much needed this year.
Abstract

Secure multiparty computation is a subfield of cryptography, which allows parties to jointly compute a function over their inputs while at the same time keeping these inputs private. This project provides the implementation, benchmarks and comparison of two different protocols for performing secure multiparty computation, specifically the secure summation of multiple inputs from different parties.
Contents
Introduction ......................................................................................................................... 1
Motivation ......................................................................................................................... 1
Objectives ......................................................................................................................... 1
Protocol One: PSA ............................................................................................................. 2
Overview ........................................................................................................................... 2
Algorithms Involved ......................................................................................................... 3
Setup ................................................................................................................................. 3
NoisyEnc .......................................................................................................................... 3
AggrDec ............................................................................................................................ 4
Example ............................................................................................................................. 4
Performance Characteristics .......................................................................................... 5
Communication rounds ................................................................................................. 5
Number of messages ....................................................................................................... 5
Decryption complexity .................................................................................................... 5
Encryption complexity .................................................................................................... 5
Security & collusion tolerance ....................................................................................... 6
Protocol Two: RASS ......................................................................................................... 7
Overview ........................................................................................................................... 7
Example ............................................................................................................................. 8
Performance characteristics ......................................................................................... 9
Communication rounds ................................................................................................. 9
Number of messages ....................................................................................................... 9
Decryption complexity .................................................................................................... 9
Encryption complexity .................................................................................................... 9
Security & collusion tolerance ....................................................................................... 9
What is VIFF? .................................................................................................................... 10
Important VIFF modules ................................................................................................. 10
Finite fields module (viff.field) ....................................................................................... 10
Runtime module (viff.runtime) ...................................................................................... 11
Utility functions module (viff.util) ................................................................................ 12
Important notice ............................................................................................................. 13
Implementation - PSA ...................................................................................................... 14
Psa.py ............................................................................................................................... 14
Setup() ............................................................................................................................. 14
Figures

FIGURE 1: PSA OVERVIEW (Shi, Chan, Riefel, Chow, & Song, 2011). Each participant adds noise $r_i$ to her value $x_i$ before encrypting it. The aggregator uses the capability $sk_o$ to decrypt a noisy sum, but learns nothing more. .......................................................... 2

FIGURE 2: UML CLASS DIAGRAM FOR PSA.py .......................................................... 14

FIGURE 3: UML CLASS DIAGRAM FOR RASS.py .......................................................... 20

Tables

TABLE 1 - PERFORMANCE CHARACTERISTICS OF PSA .......................................................... 5
TABLE 2 - PERFORMANCE CHARACTERISTICS OF RASS .................................................. 9
TABLE 3: NOISE ENC TIMINGS FOR 256BIT PSA .......................................................... 23
TABLE 4: NOISE ENC TIMINGS FOR 512BIT PSA .......................................................... 24
TABLE 5: NOISE ENC TIMINGS FOR 1024BIT PSA .......................................................... 25
TABLE 6: NAIVE AGGRDEC TIMINGS FOR 256BIT PSA .................................................. 26
TABLE 7: AGGRDEC TIMINGS FOR 256BIT PSA .......................................................... 27
TABLE 8: AGGRDEC TIMINGS FOR 512BIT PSA .......................................................... 28
TABLE 9: AGGRDEC TIMINGS FOR 1024BIT PSA .......................................................... 29
TABLE 10: SIMULATED RASS ROUND TIMINGS FOR 20-BIT MESSAGE SPACE ................. 32
TABLE 11: COMPARISON OF RASS 20MS WITH 1024BIT PSA IN A 20-BIT MESSAGE SPACE .................................................. 33
TABLE 12: RASS 50MS VS PSA 1024BIT IN A 20-BIT MESSAGE SPACE .................................. 34
Introduction

This project presents an implementation and comparison of two different secure multiparty computation protocols to allow for secure summation of multiple private inputs. A secure multiparty computation protocol allows parties to jointly compute a function over their inputs, while allowing their inputs to remain private. This insurance of privacy can be much sought-after when dealing with various pieces of sensitive or personal information, for example a voting system where the specifics of any single vote are not meant to be made public, bar the candidate that the vote is for. Another example, and the de facto example used when explaining secure multiparty computation, is that of two millionaires who wish to find out which of them is the richest, but neither wishes to disclose their net worth to the other. This example was initially suggested by Andrew C. Yao in a paper (Yao, 1982) and has come to be known as the millionaire problem.

Motivation

There are many different protocols for performing secure multiparty computation, and usually these protocols vary in their different characteristics. Different functions require specific protocols, for example in the millionaires problem the function to be computed is that of $\text{worth}_A > \text{worth}_B$ meaning we are comparing two inputs and calculating which is larger. Clearly, a protocol for summing multiple inputs is not appropriate here.

As well as differing in their function, the various protocols can differ wildly in terms of their performance. One protocol may be computationally simple, but require a large amount of communication between parties. Another may prove very computationally expensive when the size of the message space is large, but otherwise performs relatively well.

Clearly, knowing how different protocols perform in different situations would be very valuable when looking to apply secure multiparty computation as efficiently as possible. The results of this project will ideally be used in aid of a PhD project currently being undertaken in Trinity College Dublin, which aims to allow for the efficient selection of protocols at runtime based on a number of metrics e.g. network latency between parties, and the computational power available to each party.

Objectives

This project hopes to achieve the following:

- Implement two different but semantically equivalent protocols. (i.e. they both allow the computation of the same function using different methods)
- Compare these protocols based on different metrics.
- Benchmark both protocols.
- Conclude with an analysis of the results.
Protocol One: PSA

The first protocol chosen for use in this project is known as Private Stream Aggregation, or PSA. This protocol was first presented at the Network and Distributed System Security Symposium by Elaine Shi (Shi, Chan, Rieffel, Chow, & Song, 2011). It allows for the secure summation of inputs by a single untrusted aggregator party. Here we provide an overview of the main points from the paper to give a general understanding of the design and operation of the protocol.

Overview

This protocol involves a single untrusted aggregator as well as several trusted parties who provide inputs to the aggregator. It is designed in such a way that it is Aggregator oblivious, i.e. the aggregator cannot learn any unintended information about any of the parties’ inputs other than the sum of all the inputs.

The diagram below provides an overview of the protocol.

![Diagram of PSA Protocol](image)

Figure 1: PSA Overview (Shi, Chan, Rieffel, Chow, & Song, 2011). Each participant adds noise $r_i$ to her value $x_i$ before encrypting it. The aggregator uses the capability $sk_0$ to decrypt a noisy sum, but learns nothing more.

To begin with, a setup stage must occur in which a trusted dealer generates $n + 1$ random secret keys and deals them out to the different parties. The aggregator receives the capability, $sk_0$, and participant $i$ receives $sk_i$. This setup stage need only occur once for a given group of aggregator + parties, following which any number of secure summations can be made with the same set of secret keys.

For each summation, every party encrypts their data, $x_i$, after first adding some random noise $r_i$ to it. The exact encryption method will be explained later. The party’s encrypted ciphertext is referred to as $c_i$. Each party then sends their ciphertext to the aggregator, who uses the capability $sk_0$ to decrypt the sum of all the inputs.
Algorithms Involved

The following explanations of the various algorithms/stages used in PSA are based on section 5.2 of the paper (Shi, Chan, Rieffel, Chow, & Song, 2011).

We let $G$ represent a cyclic group of prime order $p^1$. We use $H$ to denote some hash function mapping integers to elements of $G$, modelled as a random oracle$^2$.

Setup
This stage is our initial setup process, and needs only be computed once for every group of players provided the parameters for the calculations don’t change (i.e. the message space size etc.). Some trusted dealer chooses a random generator $g$ from $G$, along with $n + 1$ random secret keys $sk_0, sk_1, ..., sk_n \in \mathbb{Z}_q$ in such a way that the sum of these secret keys, $sk_0 + sk_1 + ... + sk_n = 0$.

The public parameters are created, $param := g$, as well as any other values that all parties need to know. Lastly, the secret keys are to be dealt out in a secure manner. The aggregator obtains $sk_0$, also known as the capability, and participant $i$ obtains their secret key, $sk_i$.

NoisyEnc
This is the algorithm which deals with the encryption of inputs by the individual parties. It takes as inputs the public parameters $param$, the secret key of the participant $sk_i$, the time period $t$ and the participant’s randomised plaintext input $\hat{x}$. This is simply the participant’s input $x$, plus some random “noise” $r$, all modulo the prime $p$.

In order for the participant to actually encrypt their value $\hat{x}$ for the given time step $t$, they compute the following ciphertext:

\[ c \leftarrow g^{\hat{x}} \cdot H(t)^{sk_i} \]

Once this is done, the participant can send their ciphertext to the aggregator for summation without risk of their plaintext input being revealed.

---

$^1$ A cyclic group is a group containing some element $g$ where every element in the group can be represented as a power of $g$, i.e. there exists some $g$ where every element in the group can be represented by $g^n$ for some integer $n$.

$^2$ A random oracle can be thought of as a black box that responds to queries with a “random” response that is chosen uniformly from its output domain. However, the random oracle will respond the same way to subsequent identical queries.
AggrDec
This algorithm is performed by the aggregator in order to decrypt the series of ciphertexts it has received from the different parties. The inputs it takes are the public parameters param, the capability sk₀, the time step t and the list of ciphertexts, c₁, c₂, ... cₙ.

Once the aggregator has all of these, they can compute the following:

\[ V \leftarrow H(t)^{sk₀} \prod_{i=1}^{n} c_i \]

This value V can be observed to be of the form:

\[ V = g^{\sum_{i=1}^{n} \tilde{x}_i} \]

That is, V is equal to g raised to the power of the sum of the noisy inputs. In order to decrypt the sum of the inputs, it is required to compute the discrete logarithm\(^3\) of V base g, or \( \log_g V \). Herein lies the computational complexity of the PSA algorithm. Computing the discrete logarithm is a known difficult task, and the hardness of its computation has resulted in it being the basis for several public-key cryptosystems, for example DSS.

There are several algorithms for the computation of discrete logarithms, ranging from a naive brute-force approach to the much more efficient Pollard’s Lambda algorithm. The naïve algorithm requires a running time which is exponential with regard to the number of digits of the group’s size, whereas Pollard’s Lambda algorithm has a time complexity of \( O(\sqrt{b-a}) \) where b and a represent the upper and lower bounds of the range in which we search for the logarithm. In this case, we will generally be searching the entire message space, so it will be of \( O(\sqrt{\Delta}) \) where delta \( \Delta \) is the upper bound of the message space \([0..\Delta]\).

Example
For this example scenario, let us assume that there are 13 participants, as well as a separate untrusted aggregator and a trusted third party dealer.

- The trusted dealer performs the Setup algorithm.
  - A generator g is chosen from the cyclic group G. This is encapsulated in the public parameters params, which is dealt to everyone.
  - \( n + 1 \) secret keys are generated: sk₀ ... skₙ.
  - The aggregator is dealt the capability, sk₀.
  - The parties are dealt their respective secret keys, sk₁ ... skₙ.
- Each party then adds random noise to their input, before encrypting it using the NoisyEnc algorithm.
  - The encrypted ciphertext c is sent to the aggregator.

---

\(^3\) A discrete logarithm is equivalent to an ordinary logarithm, but over the elements of a finite cyclic group.
• Once the aggregator has received a ciphertext from every party, they then compute the noisy sum of the inputs using the $AggrDec$ algorithm.

Performance Characteristics

The below table shows a series of theoretical performance characteristics for the PSA protocol. Actual performance as per this project’s implementation will be discussed in a later section.

<table>
<thead>
<tr>
<th>Communication Rounds</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Messages</td>
<td>n</td>
</tr>
<tr>
<td>Decryptional complexity w.r.t message space size</td>
<td>Exponential increase in complexity with increasing message space size.</td>
</tr>
<tr>
<td>Computational complexity of encryption</td>
<td>1 hash + 2 modular exponentiations per node.</td>
</tr>
</tbody>
</table>

*Table 1 - Performance characteristics of PSA*

Communication rounds

In PSA, there is a single communication round per summation. Every participant encrypts their input, and sends it to the aggregator.

Number of messages

During each communication round, each participant sends a single message to the aggregator, resulting in $n$ messages per round.

Decryption complexity

The aggregator must perform a discrete logarithm on the product of the ciphertexts received from the participants. This is an expensive operation that scales with the square root of the plaintext sum. The cost of this step in the computation depends on the method by which the discrete logarithm is computed.

For a naïve brute-force approach, the complexity of decrypting the sum of the inputs is exponential with regard to the size of the message space. A more efficient algorithm, such as Pollard’s lambda algorithm, reduces this complexity to $O(\sqrt{\Delta})$ where $\Delta$ is the upper limit on the message space.

Encryption complexity

Each node must perform a hash, along with two modular exponentiations. Of course, the complexity here depends on the hash function used, as well as the size of the message space, however modular exponentiation is seen as quite easy to compute. In fact, modular exponentiation vs. discrete logarithms are seen as an easy one-way function, such is the difference in difficulty in calculating one over the other.
Security & collusion tolerance

Private Stream Aggregation is tolerant of any amount of collusion between parties, up to the point where all but one parties collude. I.e. its collusion tolerance is \( n - 1 \). This is due to the fact that the only information that any party learns is the final sum. In order to learn anything about the inputs of another party, it would be necessary to already know the inputs for every party but the target, and due to the noise added to each party’s inputs then only the noisy input for the target would be revealed.

In terms of actual security, PSA is only “computationally secure”. What this means is that given unbounded resources (time & computing power enough to calculate arbitrary discrete logarithms) an attacker could eventually decrypt the inputs without any collusion necessary.
Protocol Two: RASS

The second protocol examined in this project is called Ring Additive Shared Secrets. There are many different variations on this protocol, but the one that is focused on here is a ring-based approach for summation of several inputs. An overview of the protocol follows.

Overview

There are \( n \) participants, one of which is selected as the co-ordinator. It is the coordinator’s job to begin the sharing of secrets around the ring in each communication round, and it is the coordinator who will eventually receive the sum of every participant’s inputs.

Each participant has a value \( s \) which is their input to the calculation.

\[ \text{num} = \left( \frac{n+5}{6} \right) \]

shares are prepared by each participant, where division here represents integral division. E.g. for \( n = 13 \) we have \( \frac{13+5}{6} = 3 \) shares prepared by each participant.

Shares are prepared as follows:

\[ r_1 = \text{rand}; r_2 = \text{rand}; \ldots; r_{\text{num}} = s - r_1 - r_2 - \ldots; \]

The exception to this is the co-ordinator, who must prepare \( \text{num} + 1 \) shares. In this case, \( r_{\text{num}} = \text{rand}; r_{\text{num}+1} = s - r_1 - r_2 - \ldots \)

Here, the randoms must be “uniform” This means that if the values are 32 bit integers, then each integer must have a \( 1/(2^{32}) \) chance of being selected. We can see that the last share is the difference between the sum of all other shares, and the participant’s value itself. When a uniform random is added to another value modulo the domain of the random, the result is also uniformly distributed, and hence leaks no information about the value on its own.

The co-ordinator sends a share to participant one (P1) for sending to P2, P3, etc. A share is also sent to P2 for sending to P4, P6, etc. The same happens for P3, P6 and so on.

Each participants, on receipt of each share or sum of shares for subsequent participants, adds a share to the total and forwards it to the next in the chain, until it has reached all participants. At this point the total is sent back to the co-ordinator.

The co-ordinator adds all returned sums of shares, adds their final share (to prevent their value being described by the \( (n + 5)/6 \) original shares – i.e. below the collusion threshold) and publishes the result.
Example
As an example scenario, let us assume that there are 13 participants, and participant 1 has been selected as co-ordinator.

- As above, we see that \( \text{num} = \frac{(13+5)}{6} = 3. \)
- The participants have their values \( s_1, s_2, s_3 \ldots s_{13}. \)
- The first (random) shares are generated:
  - \( r_{11}, r_{12}, r_{13} \ldots r_{113}. \)
  - Each participant has an \( r_1 \) value.
- The second (also random) shares are generated:
  - \( r_{21}, r_{22}, r_{23} \ldots r_{213}. \)
  - Each participant also has an \( r_2 \) value.
- The third and final shares are generated:
  - \( r_{32} = s_2 - r_{22} - r_{12} \ldots \)
  - \( r_{313} = s_{13} - r_{213} - r_{113} \)
  - We can see from this that each final share is calculated so that \( r_1 + r_2 + r_3 = s. \)
- The co-ordinator is the exception to the above, as they must create a fourth share. In this case, \( r_{31} \) is also random, and \( r_{41} = s_1 - r_{31} - r_{21} - r_{11}. \)
  - This is to prevent the co-ordinator from having to reveal any information on their input \( s \) on any initial step of a communication round.
- Communication rounds begin.
  - P1 sends \( r_{11} \) to P2.
    - P2 sends \( r_{11} + r_{12} \) to P3.
  - P1 sends \( r_{21} \) to P3.
    - P3 sends \( r_{21} + r_{23} \) to P5.
  - P1 sends \( r_{31} \) to P4.
    - P4 sends \( r_{31} + r_{34} \) to P7.
- Once each sum has reached the last participant in the ring, they are sent back to the co-ordinator.
  - P13 sends the first round sum to P1.
  - P12 sends the second round sum to P1.
  - P11 sends the third round sum to P1.
- P1 then sums the three sums, adds their final share \( r_{41} \), and publishes the result.
- The result is \( s_1 + s_2 + s_3 \ldots + s_{13}. \)
Performance characteristics
The below table shows a series of theoretical performance characteristics for the RASS protocol. Actual performance as per this project’s implementation will be discussed in a later section.

<table>
<thead>
<tr>
<th>Communication Rounds</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Messages</td>
<td>$n \times \left(\frac{n + 5}{6}\right)$</td>
</tr>
<tr>
<td>Decryptional complexity w.r.t message space size</td>
<td>Constant.</td>
</tr>
<tr>
<td>Computational complexity of encryption</td>
<td>$(n + 5)/6$ additions per node. $\frac{n+5}{6} - 1$ uniform random variables per node.</td>
</tr>
</tbody>
</table>

*Table 2 - Performance characteristics of RASS.*

Communication rounds
For RASS, we present a standard algorithm which requires $n$ rounds of communication. This can be reduced through optimization for a specific number of participants, but our algorithm does not attempt this.

Number of messages
Our algorithm for RASS requires $n \times \left(\frac{n+5}{6}\right)$ messages. This scales with $O(n^2)$.

Decryption complexity
The decryption of the final sum is a simple side effect of the “encryption” process. That is, the final sum is revealed to be the value remaining once the last share is added to all the other sums. The operation of modulo addition is a very cheap one.

Encryption complexity
As above, the “encryption” of the inputs is simply a modulo addition on the sums. This operation is quite cheap, especially when compared to the encryption used in PSA. The generation of the uniform random variables is not particularly expensive either, provided there exists a sufficient source of entropy for the party to use.

Security & collusion tolerance
This protocol is said to be *information-theoretically* secure. What this means is that given an infinite amount of both time and computing power, there is no way for an attacker to gain any unintended information beyond what they know already.

This security comes from the uniformity of the shares of the data, and the number of shares. Even a computationally unbounded adversary must determine the value of all of a participant’s shares in order to reconstruct their original value, learning nothing until they have the last share. This means that they must corrupt every participant with whom communication has taken place. This can be set as high as $n - 1$, but in order to fulfill the standard collusion tolerance requirement in SMC of $t < n/3$, in our algorithm this is $2 \times \left(\frac{n+5}{6}\right)$ using integer division.
VIFF: The Virtual Ideal Functionality Framework

What is VIFF?

VIFF is a Python framework that allows you to specify and perform secure multiparty computations cleanly and easily. It provides many useful classes and functions for performing various cryptographic operations, sharing of secrets, automatic parallel execution along with much more. It is the foundation upon which this project’s implementation is built.

VIFF itself is built on top of several popular and well-established Python frameworks and libraries.

- The OpenSSL⁴ & PyOpenSSL⁵ libraries provide the basis for most of VIFF’s cryptography related functionality.
- GMPY⁶ is a Python extension module, allowing the use of the GMP library from the Python language. GMP provides fast multiple-precision integer and rational arithmetic. It is used as the underlying implementation for much of VIFF’s mathematical functionality.
- Twisted⁷ is “an event-driven networking engine written in Python” and VIFF uses it under the hood to provide its various communication functions.

The version of VIFF used in this project is VIFF-1.0., released December 14th, 2009.

Important VIFF modules

Code snippets below taken from the documentation pages for VIFF⁸

Finite fields module (viff.field⁹)

This module contains classes that model the functionality of Galois (or finite) fields. A field is a concept from abstract algebra. It is simply a set, together with the two binary operations of addition and multiplication. A finite field is a field which contains a finite number of elements.

An example of a field is the field of all real numbers. An example of a finite field is the field of integers modulo a prime p, as seen in use in the PSA protocol.

This is possibly the module which provides the most utility to this project, as finite field arithmetic is used extensively throughout the Private Stream Aggregation protocol.

The main functionality provided in this module is the $GF$ class, which represents a finite field, and the $FieldElement$ class, which represents an element from a $GF$.

⁴ http://www.openssl.org/
⁵ http://pyopenssl.sourceforge.net/
⁶ http://code.google.com/p/gmpy/
⁷ http://twistedmatrix.com/
⁸ http://viff.dk/doc/index.html
⁹ http://viff.dk/doc/field.html
Example: Defining a field with modulus p (p must be prime):

```python
>>> Zp = GF(p)
```

We can now access elements from this field as such:

```python
>>> a = Zp(12)
>>> b = Zp(6)
>>> c = Zp(2)
```

All of the standard arithmetic operations (addition, subtraction, multiplication, division) as well as some other operations such as exponentiation are defined on FieldElement types. FieldElements from different fields cannot be used together in arithmetic operations.

In particular, the exponentiation implemented for FieldElements makes use of a technique called square-and-multiply, i.e. fast modular exponentiation, to greatly speed up exponentiation of FieldElement bases. This is incredibly useful for this project, as we are dealing with quite large primes with hundreds of digits.

Runtime module (viff.runtime\(^{10}\))

Contained in this module is the “ideal functionality” that gives VIFF its name. The Runtime class is responsible for handling the communication between parties, sharing their inputs between them and running the various calculations on their inputs.

There are various different subclasses of Runtime shipped with VIFF, each providing a runtime that handles the communication and sharing of inputs differently. For example, the viff.passive.PassiveRuntime class provides a Runtime that supports sharing of inputs via two different SMC techniques, Shamir’s Secret Sharing (implemented using the functions in the viff.shamir\(^{11}\) module) and Pseudo-Random Secret Sharing (viff.prss\(^{12}\) module).

Neither of these methods are appropriate for use in PSA or RASS, but they are what VIFF provides “out of the box”. PSA and RASS do not require as complex a mechanism for sharing inputs as either Shamir sharing or PRSS, they simply require the ability to send data directly to a single party.

In order for this to be implemented within VIFF, one would need to subclass Runtime, as well as provide a custom version of the ShareExchanger class used within the Runtime. This ShareExchanger class is a subclass of the Twisted frameworks’ Protocol class, and represents a channel of communication between two endpoints. In the PassiveRuntime class, there is an instance of a ShareExchanger for every player involved in the protocol. A special version of the ShareExchanger is used in the list of players, in the index that represents the current player, called SelfShareExchanger, and provides dummy functions for “sharing” an input with itself.

---

\(^{10}\) [http://viff.dk/doc/runtime.html](http://viff.dk/doc/runtime.html)

\(^{11}\) [http://viff.dk/doc/shamir.html](http://viff.dk/doc/shamir.html)

\(^{12}\) [http://viff.dk/doc/prss.html](http://viff.dk/doc/prss.html)
Utility functions module (viff.util\[^{13}\])

The functions contained in this module provide a variety of utilities, from the random number generator used throughout the VIFF modules themselves, to various profiling-related functions. Most importantly from the point of view of this project, is the \texttt{find_random_prime(k)} function. This function returns a random prime number with up to $k$ significant bits, a thoroughly useful function for implementing the PSA protocol.

\[^{13}\] \url{http://viff.dk/doc/util.html}
Implementation

Important notice

Important to note is that these implementations are a black-box, and all benchmarks based on them are treated as such. What is meant by this is that there is no actual distributed communication between parties currently in place. Instead, all benchmarks are done based on the different computational steps involved in each protocol.

The effects of the communication aspects of the different protocols can still be somewhat accurately measured and estimated, as the effects of latency are quite straightforward (a simple delay is introduced upon any message being sent between parties) and other aspects such as bandwidth requirement can be measured without needing to simulate the communication.
Implementation - PSA

This section will discuss my approach to and the resulting code for implementing the PSA protocol using Python and VIFF.

Psa.py

This file contains the implementation of the base PSA functionality. The three main algorithms (Setup, NoisyEnc and AggrDec) are found here. All three algorithms are implemented as member functions in the PSA class. The public parameters are encapsulated in a Params class. A thorough analysis of the code follows.

Setup()

Firstly there is the setup algorithm. In our current implementation of this algorithm, the trusted dealer aspect is ignored, and the relevant values are simply returned by the setup() function.

```python
def setup(self, n, t, delta, k, p = 0):
    rand = random.SystemRandom()
    sigma = delta / math.sqrt(n*(1 - t))
    
    The arguments to this function should be sufficiently explained by the comments above its prototype.

The first thing this function does is to create an instance of the random.SystemRandom class, which is a cryptographically secure (i.e. non-deterministic) random number generator which uses an appropriate source of randomness depending on the host OS. We then calculate a value for sigma, used later to generate our random noise before encryption of an input.

rand = random.SystemRandom()
sigma = delta / math.sqrt(n*(1 - t))
```

The next step is to calculate a suitable prime \( p \), if one hasn’t been provided for us as an
argument. This makes use of a private function of the PSA class, `_find_p(k)` which generates what is known as a safe prime, i.e. a prime number of the form $2q + 1$ where $q$ is also prime. Safe primes are desirable in cryptography as they are more resilient to attacks. The function returns both $p$ and $q$. $p$ is found using a naïve approach, generating a random prime using the viff.util.find_random_prime(k) function to randomly find primes of the right bit length, and simply check whether it meets the above requirements for a safe prime. If not, we generate a new prime and check again. As a result of this randomness, the running time for the setup algorithm can vary wildly, which is why the ability to specify a value for $p$ is given. The GMPY library’s mpz class represents arbitrary precision integers, and provides a handy primality check in the form of is_prime() which is used in our search for $p$.

```python
if p == 0:
    q, p = self._find_p(k)
else:
    q = (p - 1) / 2
assert n*delta < q
...

def _find_p(self, k):
    q = find_random_prime(k)
    while not mpz(2*q + 1).is_prime():
        q = find_random_prime(k)
    return (q, 2*q + 1)
```

Next, we create our fields of integers mod $p$ and mod $q$, $Z_p$ and $Z_q$, using the viff.field.GF class:

```python
Zp = GF(p)
Zq = GF(q)
```

The next task is to find our generator, $g$. Due to our modulus $p$ being a safe prime, finding a generator becomes a simple enough algorithm:

```python
self.g = self._find_gen(Zp, q, rand) ** 2
...

def _find_gen(self, Zp, q, rand):
    g = Zp(rand.randrange(Zp.modulus))
    while g^2 == 1 or g**q == 1:
        g = Zp(rand.randrange(Zp.modulus))
    return g
```

The last step is to generate our secret keys and capability, then return these along with a Params class encapsulating the values used in the NoisyEnc and AggrDec algorithms.
tmpsks = [0] * (n + 1)
self.sks = []
for sk in tmpsks:
    self.sks.append(Zq(sk))
for i in range(1, n + 1):
    self.sks[i] = Zq(rand.randint(0, Zq.modulus))
self.sks[0] = -reduce(lambda x, y: x + y, self.sks[1:])
assert sum(self.sks) == 0
self.sks = map(lambda x: x.unsigned(), self.sks)

return (self.Params(Zp, self.g, sigma, delta, rand, Zq), self.sks)

This covers the entirety of the setup algorithm’s implementation.
NoisyEnc()
Next we will review the implementation of the encryption algorithm, NoisyEnc.

The paper (Shi, Chan, Rieffel, Chow, & Song, 2011) gives the following equation for computing the encrypted ciphertext:

\[ c \leftarrow g^{\hat{x}} \ast H(t)^{sk_t} \]

# params is the public parameters, an instance of class Params
# sk is our personal secret key
# t is the timestep for which we are computing this encrypted input
# x is our plaintext, unmodified input
def NoisyEnc(self, params, sk, t, x):
    
g = params.g
H = params.H
Zp = params.Zp
sigma = params.sigma
rand = params.rand

Next, we generate our random noise using the Gaussian function with a std. deviation of sigma and add it to our input to produce \( \hat{x} \):

r = int(rand.gauss(0, sigma))
xbar = Zp(x + r).unsigned()

We then compute \( g^{\hat{x}} \) as per the paper (Shi, Chan, Rieffel, Chow, & Song, 2011).

gxbar = (g**(xbar))

Now that we have \( g^{\hat{x}} \), we can multiply this by \( H(t)^{sk} \) in order to calculate our encrypted ciphertext. The same caveat applies regarding a possible negative exponent. The ciphertext is returned, along with the plaintext input and the added noise to allow for the testing of AggrDec:

\[ c = gxbar \ast (H(t)^{sk}) \]

return (c, x, r)

This concludes the code analysis for the NoisyEnc algorithm.
The final piece of functionality in the PSA class is the AggrDec() function, which implements the algorithm of the same name. The aim here is to first compute:

\[ V \leftarrow H(t)^{sk_0} \prod_{i=1}^{n} c_i \]

Recall that \( V \) is of the form \( V = g^{\sum_{i=1}^{n} \hat{x}_i} \) so to compute the unencrypted sum, we must compute the discrete logarithm of \( V \) base \( g \).

```python
# params is the public parameters
# sk is the capability, sk0
# t is the timestep
# cs is a list of the encrypted ciphertexts
def AggrDec(self, params, sk, t, cs):
    H = params.H
    g = params.g
    Zp = params.Zp
    delta = params.delta  # size of the message space

    cprod = reduce(lambda x, y: x * y, cs, 1)  # Get the product of all the ciphertexts

    V is calculated:
    v = (H(t)**sk) * cprod

    Now that we have \( V \), the final step is to calculate the discrete logarithm of \( V \) base \( g \). As mentioned in the protocol overview, there are several approaches to this calculation. This project originally used the naïve brute-force approach, which was implemented as such:

    h = g
    for x in longrange(len(cs)*delta):
        if v == h:
            return x + 1
        h *= g

    return None
```

This algorithm simply raises \( g \) to higher and higher powers until either the resulting value equals \( V \), or we raise it to a power past some upper bound. In this case, our bound is \( n * \delta \).
Later on in the project an implementation of Pollard’s lambda algorithm was adapted from the open-source mathematical framework for Python, Sage\textsuperscript{14,15}. Once this was correctly ported from using the Sage libraries to those providing equal or similar functionality in VIFF, GMPY etc. a substantial speedup was gained in the overall running time of the decryption algorithm. The exact implementation of pollard’s lambda won’t be discussed here, as it wasn’t written as part of this project, merely adapted from existing code.

```python
try:
    assert type(v) == type(g)
    x = discrete_log_lambda(v, g, (0, len(cs)*delta))
except ValueError:
    print "Pollard-lambda failed to find log with standard operator, trying alternative"
    x = discrete_log_lambda(v, g, (0, len(cs)*delta), '+')
return x
```

The discrete_log_lambda() function is passed our value, \( V \), the base \( g \), and a tuple containing a lower and an upper bound within which to search. You can optionally specify for the function to use alternative operators during its calculations, this is done here when using the standard operators fail to find a logarithm.

And that concludes the code analysis for each of the three main algorithms in the PSA protocol.

\textsuperscript{14} http://www.sagemath.org/
\textsuperscript{15} http://trac.sagemath.org/sage_trac/attachment/ticket/5112/trac-5112-pollard_review.patch
Implementation – RASS

As RASS is a much simpler protocol than PSA, from a computation point of view, there is much less code to discuss. As of this implementation, and the fact that there isn’t any full simulation of communication between parties, the only functionality to be implemented was the generation of the shares. Everything else is simple modulo arithmetic and sending/receiving of shares between parties.

Rass.py

The rass.py file contains the implementation of the RASS class. This class provides a helper function to generate a series of shares based on the parameters given to the class upon instantiation.

![Figure 3: UML Class diagram for Rass.py](image)

**gen_shares()**

This function generates a list of shares for a party as described in the RASS overview section. The shares are constructed per the RASS algorithm, generating \( \text{num} \) shares, or \( \text{num} + 1 \) in the case where \( \text{self.id} = 1 \) (i.e. we assume participant one is the coordinator).

For the first \( \text{num} - 1 \) shares, each share is generated thusly:

```python
rs = []
for i in range(1, self.numShares + 2):
    if i < self.numShares or (self.id == 1 and i < self.numShares + 1):
        rs.append(self.rand.randint(0, self.maxVal))
```

The final share is computed like so, and then the list of shares is returned:

```python
elif i == self.numShares or (self.id == 1 and i == self.numShares + 1):
    rs.append(self.s - reduce(self.modAdd, rs[:i-1]))
self.rs = rs
return rs
```
And that is the extent of the helper code written for the RASS protocol. It is clear to see that the computational complexity of the PSA and RASS protocols are vastly different. All that is left to do in order to perform a computation with the RASS protocol is some modular additions and simple sending/receiving of values to appropriate parties, neither of which are complex tasks.
Benchmarks

A set of benchmarks were performed and the results tallied and graphed as part of the project. These benchmarks attempt to look at the important measurable performance characteristics common to both protocols, gather data regarding this project’s implementations regarding these characteristics, and analyze them and how they might be affected by different runtime variables such as processing power, latency to other participants etc.

These benchmarks were run on a mid-high end PC, the relevant specs and software versions of which are described in the table below and should be taken into account when assessing the results.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i5-3570K @ 4.10 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>8GB DDR3-1600 @ 1333MHz</td>
</tr>
<tr>
<td>Operating System</td>
<td>Windows 8 x64</td>
</tr>
<tr>
<td>Python version</td>
<td>2.7.3 (32-bit)</td>
</tr>
<tr>
<td>VIFF version</td>
<td>1.0</td>
</tr>
<tr>
<td>GMPY version</td>
<td>1.15</td>
</tr>
<tr>
<td>Twisted version</td>
<td>12.3.0</td>
</tr>
</tbody>
</table>

PSA

Time taken to encrypt + decrypt.

Firstly, timing data was collected measuring the running time of the NoisyEnc() function and the AggrDec() function, given appropriate valid arguments for varying values of \(n, b \& k\). As a reminder, these represent the number of participants, the number of significant bits in the message space, and the security parameter (the number of significant bits used to generate the prime numbers \(p \& q\)).

This benchmark was run while varying the value of \(b\) in the range \([8, 10, \ldots, 20]\), the value of \(n\) in the range \([4, 8, 16, 32, 64]\) and with \(k\) in the range \([256, 512, 1024]\). Three sets of data and an accompanying graph follow, one for each value of \(k\). Each set contains timings of the NoisyEnc() function for each combination of \(n\) and \(b\). The timings have been averaged in order to provide smoother graphs and more accurate results.
Table 3: NoisyEnc timings for 256bit PSA

<table>
<thead>
<tr>
<th></th>
<th>n = 4</th>
<th>n = 8</th>
<th>n = 16</th>
<th>n = 32</th>
<th>n = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.002954</td>
<td>0.003003</td>
<td>0.002855</td>
<td>0.002865</td>
<td>0.002883</td>
</tr>
<tr>
<td>10-bit</td>
<td>0.002955</td>
<td>0.002926</td>
<td>0.002906</td>
<td>0.002893</td>
<td>0.002885</td>
</tr>
<tr>
<td>12-bit</td>
<td>0.002891</td>
<td>0.002931</td>
<td>0.002892</td>
<td>0.002915</td>
<td>0.002895</td>
</tr>
<tr>
<td>14-bit</td>
<td>0.002976</td>
<td>0.002888</td>
<td>0.002932</td>
<td>0.002904</td>
<td>0.002907</td>
</tr>
<tr>
<td>16-bit</td>
<td>0.003034</td>
<td>0.002997</td>
<td>0.00291</td>
<td>0.002865</td>
<td>0.002888</td>
</tr>
<tr>
<td>18-bit</td>
<td>0.003027</td>
<td>0.002944</td>
<td>0.002964</td>
<td>0.00291</td>
<td>0.002876</td>
</tr>
<tr>
<td>20-bit</td>
<td>0.002981</td>
<td>0.002921</td>
<td>0.00295</td>
<td>0.002901</td>
<td>0.002896</td>
</tr>
</tbody>
</table>

What we can take from this graph is that there is little to no variation in encryption time, neither with an increasing message space size nor an increasing number of parties. Of course, the number of parties would intuitively not have had any effect on the encryption time, since it is party-independent, but the fact that we see no increase in encryption time with a larger message space is useful information.

Summary of observations from above information:
- Encryption time is apparently independent of message space size and number of players.
**Table 4: NoisyEnc timings for 512bit PSA**

<table>
<thead>
<tr>
<th></th>
<th>n = 4</th>
<th>n = 8</th>
<th>n = 16</th>
<th>n = 32</th>
<th>n = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.007196</td>
<td>0.006807</td>
<td>0.007014</td>
<td>0.006705</td>
<td>0.00682</td>
</tr>
<tr>
<td>10-bit</td>
<td>0.00658</td>
<td>0.006925</td>
<td>0.007161</td>
<td>0.006713</td>
<td>0.006746</td>
</tr>
<tr>
<td>12-bit</td>
<td>0.00816</td>
<td>0.006593</td>
<td>0.006916</td>
<td>0.007039</td>
<td>0.006802</td>
</tr>
<tr>
<td>14-bit</td>
<td>0.007432</td>
<td>0.007387</td>
<td>0.006833</td>
<td>0.006784</td>
<td>0.006829</td>
</tr>
<tr>
<td>16-bit</td>
<td>0.006639</td>
<td>0.006913</td>
<td>0.006926</td>
<td>0.006795</td>
<td>0.00679</td>
</tr>
<tr>
<td>18-bit</td>
<td>0.007811</td>
<td>0.006626</td>
<td>0.006677</td>
<td>0.00678</td>
<td>0.006935</td>
</tr>
<tr>
<td>20-bit</td>
<td>0.007289</td>
<td>0.00667</td>
<td>0.007213</td>
<td>0.007037</td>
<td>0.006821</td>
</tr>
</tbody>
</table>

We now move from 256-bit to 512-bit PSA, i.e. $k = 512$. It can be observed from the timings and the graph that a similar behavior appears as was seen in 256-bit PSA, wherein the timings all stay within the same range independent of the values for $n$ and $b$. However, we also see that the average time taken to encrypt a value has increased, from an average of 0.00292 seconds to 0.00695 seconds. Clearly the increase in size of our security parameter, and thus the primes $p$ and $q$ which define the size of our integral fields $\mathbb{Z}_p$ & $\mathbb{Z}_q$, has had a noticeable impact on the performance of this algorithm.

Summary of observations from above information:
- As with 256-bit PSA, an increase in either $b$ or $n$ has little to no impact on the running time of the encryption function.
- However, the increase of $k$ from 256 to 512 has caused the average encryption time to approximately double, going from 0.00292 seconds to 0.00695 seconds, an increase of ~138%.
Once again, we observe that changes in $b$ and $n$ have no impact on the encryption time. And the change from $k = 512$ to $k = 1024$ has brought yet another increase in encryption time. This time it is much more significant, going from an average of 0.00695 seconds to an average of 0.03296 seconds, an increase of ~374%. We can see that the initial doubling of the security parameter from 256 to 512 caused an approximate doubling in computation cost, and a further doubling of the security parameter caused an approximate further quadrupling in the computation cost. This displays an exponential increase in cost as the security parameter increases.

Summary of observations from above information:
- Increase from 512 to 1024 bit security parameter caused an approximate 374% increase in computation time.
- The % increase in computation time is double what it was for the previous step, implying exponential increase in cost as $k$ increases.
We now begin to look at the performance of the AggrDec() function in PSA, our ciphertext decryption algorithm. The same three sets of variables were used to collect data for this algorithm, as well as a single set collected using an implementation of AggrDec() which computes the discrete logarithm via the naïve approach, instead of the much more efficient Pollard’s lambda approach. The data for this brute-force version is show below.

Table 6: naïve AggrDec timings for 256bit PSA

<table>
<thead>
<tr>
<th></th>
<th>n = 4</th>
<th>n = 8</th>
<th>n = 16</th>
<th>n = 32</th>
<th>n = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.032863078</td>
<td>0.041459634</td>
<td>0.073417012</td>
<td>0.121898019</td>
<td>0.228201789</td>
</tr>
<tr>
<td>10-bit</td>
<td>0.046605581</td>
<td>0.181806408</td>
<td>0.236544585</td>
<td>0.443869359</td>
<td>0.953419853</td>
</tr>
<tr>
<td>12-bit</td>
<td>0.467369048</td>
<td>0.764561668</td>
<td>0.860476776</td>
<td>1.862728749</td>
<td>3.676138324</td>
</tr>
<tr>
<td>14-bit</td>
<td>1.145156226</td>
<td>1.842903317</td>
<td>2.650277323</td>
<td>7.390099153</td>
<td>14.4845031</td>
</tr>
<tr>
<td>16-bit</td>
<td>1.006737209</td>
<td>7.918598029</td>
<td>17.63213846</td>
<td>23.87106394</td>
<td>61.54712099</td>
</tr>
<tr>
<td>18-bit</td>
<td>21.16592215</td>
<td>26.1127001</td>
<td>58.96195034</td>
<td>105.9132986</td>
<td>264.0192355</td>
</tr>
<tr>
<td>20-bit</td>
<td>4.703640578</td>
<td>68.03016907</td>
<td>305.5875995</td>
<td>441.9151389</td>
<td>893.5151513</td>
</tr>
</tbody>
</table>

Unfortunately there wasn’t an obvious way to show the data on this graph in a more effective manner. The sheer inefficiency of this method causes the timings for 8, 10 and 12-bit message spaces to not be visible on the graph as the huge increase in decryption time as the size of the message space increases hides such small values. As we move from a 16-bit message space to an 18-bit message space, the decryption time approximately quadruples, and again as we move from 18-bits to 20-bits. Of course, such inefficiency is to be expected when using a brute-force approach to calculating the discrete logarithm, as it is just not feasible to do so except for tiny message spaces.
Summary of observations from above information:

- Brute force approach to calculating discrete logarithms is not feasible for anything but a very small message space.
- Cost increase is vast, exponential as the message space increases.

Now the same values for $k$, $b$ & $n$ will be used with a more efficient method of computing the discrete logarithm, Pollard’s Lambda algorithm.

Table 7: AggrDec timings for 256bit PSA

<table>
<thead>
<tr>
<th></th>
<th>n = 4</th>
<th>n = 8</th>
<th>n = 16</th>
<th>n = 32</th>
<th>n = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.009195581</td>
<td>0.015530516</td>
<td>0.013073021</td>
<td>0.015083805</td>
<td>0.020619747</td>
</tr>
<tr>
<td>10-bit</td>
<td>0.015312715</td>
<td>0.023997561</td>
<td>0.017241057</td>
<td>0.033057199</td>
<td>0.080474344</td>
</tr>
<tr>
<td>12-bit</td>
<td>0.022022443</td>
<td>0.046805549</td>
<td>0.028687149</td>
<td>0.043439137</td>
<td>0.064132264</td>
</tr>
<tr>
<td>14-bit</td>
<td>0.038698387</td>
<td>0.040326193</td>
<td>0.052456969</td>
<td>0.076766467</td>
<td>0.111833573</td>
</tr>
<tr>
<td>16-bit</td>
<td>0.076658005</td>
<td>0.112047574</td>
<td>0.166254258</td>
<td>0.222583697</td>
<td>0.355008952</td>
</tr>
<tr>
<td>18-bit</td>
<td>0.123714975</td>
<td>0.17619504</td>
<td>0.220200459</td>
<td>0.313532654</td>
<td>0.410798128</td>
</tr>
<tr>
<td>20-bit</td>
<td>0.21311126</td>
<td>0.354690582</td>
<td>0.476174049</td>
<td>0.548243034</td>
<td>0.872926799</td>
</tr>
</tbody>
</table>

We can immediately see that the change from brute-force to Pollard’s lambda algorithm has yielded a substantial performance gain. The most dramatic increase is that of the 20-bit message space, which went from taking almost 900 seconds previously to under a second. We can also see that contrary to the encryption algorithm, the increases in the values of $b$ and $n$ yields a performance hit. As the number of participants $n$ increases, the running time of the decryption algorithm too increases. This is again intuitive, as the higher the value of $n$, the larger the number of encrypted ciphertexts that need decrypting, and the larger the product will be. And we can see that alongside an increase in message space size comes an increase in time taken for decryption. This follows, as
with a larger message space the bounds given to Pollard’s lambda algorithm grows, as well as simply the size of the values upon which arithmetic operations are performed.

Summary of observations from above information:
- Increases to either $b$ or $n$ will result in a performance hit for decryption.

<table>
<thead>
<tr>
<th>Table 8: AggrDec timings for 512bit PSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 4</td>
</tr>
<tr>
<td>8-bit</td>
</tr>
<tr>
<td>10-bit</td>
</tr>
<tr>
<td>12-bit</td>
</tr>
<tr>
<td>14-bit</td>
</tr>
<tr>
<td>16-bit</td>
</tr>
<tr>
<td>18-bit</td>
</tr>
<tr>
<td>20-bit</td>
</tr>
</tbody>
</table>

As with 256-bit PSA, we observe an increase is computation time with an increase in either $n$ or $b$. On top of this, the move from $k = 256$ to $k = 512$ has caused a slight increase in computation time required across the board.

Summary of observations from above information:
- An increase in $k$ results in a general performance penalty for decryption, with the average running time for a 20-bit message space increasing from 0.49302 seconds to 0.55749 seconds, an increase of ~13%.
Table 9: AggrDec timings for 1024bit PSA

<table>
<thead>
<tr>
<th>n = 4</th>
<th>n = 8</th>
<th>n = 16</th>
<th>n = 32</th>
<th>n = 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit</td>
<td>0.039372</td>
<td>0.041558</td>
<td>0.045971</td>
<td>0.049502</td>
</tr>
<tr>
<td>10-bit</td>
<td>0.055961</td>
<td>0.047146</td>
<td>0.053245</td>
<td>0.063470</td>
</tr>
<tr>
<td>12-bit</td>
<td>0.049294</td>
<td>0.062837</td>
<td>0.08052</td>
<td>0.085847</td>
</tr>
<tr>
<td>14-bit</td>
<td>0.06813</td>
<td>0.086906</td>
<td>0.206451</td>
<td>0.141584</td>
</tr>
<tr>
<td>16-bit</td>
<td>0.194531</td>
<td>0.277767</td>
<td>0.171313</td>
<td>0.309764</td>
</tr>
<tr>
<td>18-bit</td>
<td>0.21388</td>
<td>0.477878</td>
<td>0.552075</td>
<td>0.728115</td>
</tr>
<tr>
<td>20-bit</td>
<td>0.293225</td>
<td>0.353786</td>
<td>0.620368</td>
<td>0.849944</td>
</tr>
</tbody>
</table>

Lastly we have the 1024-bits security parameter timings. As with the previous sets of timings, decryption time increases with $b$ and $n$, and there is another general increase across the board with the change from 512-bit to 1024-bit security.

Summary of observations from above information:

- Another increase in decryption time, this time from an average of 0.55749 seconds for the 20-bit message space, to 0.70143 seconds. This is an increase of ~25.8%, double the percentage difference from the last increase of $k$. Thus we can tell that the computational cost increases exponentially with an increase in $k$, as with encryption.

These observations match our theoretical performance characteristics for computation given for PSA in Table 1 - Performance characteristics of PSA.
Communication costs

As was mentioned in the *Important notice* portion of the Implementation section, this project did not implement communication between parties. As a result, there are no practical benchmark results for these characteristics. However, it can be seen from a simple overview of the protocol that the theoretical characteristics given in Table 1 - Performance characteristics of PSA, hold. Thus, this section will give a brief rundown of the implications of these characteristics with regard to various potential changes in runtime conditions, as well as any relevant implications for changes in parameters as seen in the previous section.

The number of communication rounds in PSA is constant, and is always going to be 1. That is, no matter how many players are involved, the players need only send their ciphertexts to the aggregator a single time.

The number of messages to be sent, however, depends on the number of participants, $n$. This follows, as every participant has an input and needs to send that input to the aggregator. The implication of this from a communication point of view is that as $n$ increases, the amount of data the aggregator needs to receive increases linearly. If bandwidth is very limited, this aspect could come into play.

The effect of latency on the overall calculation would be mostly negligible apart from the fact that the final result can’t be calculated until the aggregator has received every party’s input. There are no chains of communication which could be delayed by large amounts of latency.

Of course if a node were to become unreachable, all progress on the computation would halt as the aggregator requires every participant’s plaintext in order to decrypt the final sum.
RASS
This section contains the benchmarks performed on the RASS protocol.

Time to encrypt/decrypt

In terms of the “encryption” for RASS, which is essentially just modulo addition, during the course of benchmarking we were unable to obtain accurate timings for this operation as there was no obvious way to get access to a timer with a high enough resolution with which to measure it. This is due to the extreme cheapness of the operation. Even with larger values for the message space size, there didn’t seem to be any way of obtaining accurate measurements for it. As a result we must conclude that the “encryption” cost of RASS is so low as to be regarded as negligible, even with a 32-bit message space, much larger than the largest space benchmarked for PSA.

The decryption is the same, and in fact is just a by-product of the final encryption phase as the coordinator adds their final share to the sums. As a result, this shows just how computationally simple the RASS protocol is. The most basic of hardware could perform the operations in under a second.

Communication costs
This is where the restrictions to RASS come in to play. Communication characteristics are the only ones where any significant scaling occurs, as we have seen from Table 2 - Performance characteristics of RASS.

Without having implemented actual communication between parties, the closest to an accurate benchmark that could be achieved was a simple set of modulo additions on a single local host, whilst simulating the latency that would be observed as one party forwards the share on to the next in chain with the time.sleep() function. As modern operating systems are not real-time, and are in fact preemptive, it is not possible to sleep for a given time with perfect accuracy. The time.sleep() function may cause the thread to sleep for more or less than the specified time, especially as lower sleep values are provided. This is due to the operating system’s scheduler not awakening the thread immediately upon having slept for the specified amount of time.

On top of that, it was decided to only simulate a single round of communication as in a real world implementation of the protocol, the round would commence in parallel, greatly reducing running time versus serial execution. This simulation thus assumes “perfect” parallel communication rounds, i.e. that all communication rounds execute exactly side-by-side, and finish all at once. This, together with the inaccuracy of the time.sleep() function, and the fact that network latency isn’t constant, means the results of the simulation are far from ideal. However they do provide some use in that they are an approximation of the real running time, and could always be improved upon in future work.
Table 10: Simulated RASS round timings for 20-bit message space

<table>
<thead>
<tr>
<th>n</th>
<th>50ms</th>
<th>250ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.403953</td>
<td>2.006395</td>
</tr>
<tr>
<td>16</td>
<td>0.807723</td>
<td>4.012684</td>
</tr>
<tr>
<td>32</td>
<td>1.614208</td>
<td>8.032382</td>
</tr>
<tr>
<td>64</td>
<td>3.274592</td>
<td>16.08217</td>
</tr>
<tr>
<td>128</td>
<td>6.568541</td>
<td>32.1777</td>
</tr>
</tbody>
</table>

As shown in the above graph and table, a five times increase in latency causes an approximately five times increase in total round time. On top of this, as the number of participants doubles, so too does the simulated round time. Thus it seems fair to say that according to our basic simulation, the computation time of RASS scales linearly with both $n$, the number of participants, as well as the average network latency between each participant.

These observations are far from ground-breaking, but they are necessary to make in order to have some data with which to make comparisons against PSA.

Summary of observations from above information:

- RASS round time increases linearly with $n$ and latency between players.
Comparison

In this section we present a comparison of the two protocols and their performance, based on the benchmarks shown in the previous two sections. To begin with, we shall present data comparing the performance of the RASS simulation with 20 milliseconds of induced latency, to the 1024-bit PSA protocol. Both protocols are operating with a 20-bit message space.

Table 11: Comparison of RASS 20ms with 1024bit PSA in a 20-bit message space

<table>
<thead>
<tr>
<th>n</th>
<th>RASS:20ms</th>
<th>PSA:1024bit Enc + Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.163356191</td>
<td>0.409926047</td>
</tr>
<tr>
<td>12</td>
<td>0.245034287</td>
<td>0.570447027</td>
</tr>
<tr>
<td>16</td>
<td>0.32781425</td>
<td>0.570227472</td>
</tr>
<tr>
<td>20</td>
<td>0.409859635</td>
<td>0.715160342</td>
</tr>
<tr>
<td>24</td>
<td>0.492088664</td>
<td>0.705557518</td>
</tr>
<tr>
<td>28</td>
<td>0.574317694</td>
<td>0.802716868</td>
</tr>
<tr>
<td>32</td>
<td>0.656546723</td>
<td>0.907972856</td>
</tr>
<tr>
<td>36</td>
<td>0.738775753</td>
<td>0.867653387</td>
</tr>
<tr>
<td>40</td>
<td>0.821004782</td>
<td>0.947846198</td>
</tr>
<tr>
<td>44</td>
<td>0.903233812</td>
<td>0.987371059</td>
</tr>
<tr>
<td>48</td>
<td>0.985462841</td>
<td>0.994064708</td>
</tr>
<tr>
<td>52</td>
<td>1.06769187</td>
<td>1.143411188</td>
</tr>
<tr>
<td>56</td>
<td>1.1499209</td>
<td>1.03250378</td>
</tr>
<tr>
<td>60</td>
<td>1.232149929</td>
<td>1.06834883</td>
</tr>
<tr>
<td>64</td>
<td>1.314378959</td>
<td>1.566840596</td>
</tr>
</tbody>
</table>

We can clearly see that the simulated RASS with 20ms of latency performs better at lower values of n, but PSA begins to catch up as n approaches 60.
Now if we compare the same PSA protocol, with a simulated RASS and 50ms of induced latency we can see PSA definitively pulling ahead.

*Table 12: RASS 50ms vs. PSA 1024bit in a 20-bit message space*

<table>
<thead>
<tr>
<th>n</th>
<th>RASS:50ms</th>
<th>PSA:1024bit Enc + Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.403953334</td>
<td>0.409926047</td>
</tr>
<tr>
<td>12</td>
<td>0.605930001</td>
<td>0.570447027</td>
</tr>
<tr>
<td>16</td>
<td>0.807723364</td>
<td>0.570227472</td>
</tr>
<tr>
<td>20</td>
<td>1.00963893</td>
<td>0.715160342</td>
</tr>
<tr>
<td>24</td>
<td>1.211523945</td>
<td>0.705557518</td>
</tr>
<tr>
<td>28</td>
<td>1.41340896</td>
<td>0.802716868</td>
</tr>
<tr>
<td>32</td>
<td>1.615293976</td>
<td>0.907972856</td>
</tr>
<tr>
<td>36</td>
<td>1.817178991</td>
<td>0.867653387</td>
</tr>
<tr>
<td>40</td>
<td>2.019064006</td>
<td>0.947846198</td>
</tr>
<tr>
<td>44</td>
<td>2.220949021</td>
<td>0.987371059</td>
</tr>
<tr>
<td>48</td>
<td>2.422834036</td>
<td>0.994064708</td>
</tr>
<tr>
<td>52</td>
<td>2.624719052</td>
<td>1.143411188</td>
</tr>
<tr>
<td>56</td>
<td>2.826604067</td>
<td>1.03250378</td>
</tr>
<tr>
<td>60</td>
<td>3.028489082</td>
<td>1.06834883</td>
</tr>
<tr>
<td>64</td>
<td>3.230374097</td>
<td>1.566840596</td>
</tr>
</tbody>
</table>

![RASS 50ms Latency vs 1024bit PSA](chart.png)
Conclusions and further work

This project has been successful in providing an implementation of the various algorithms involved in the PSA protocol, as well as providing code to assist in implementing and simulating the RASS protocol. Both were benchmarked and analysed as to their strengths and weaknesses, as well as being compared and contrasted with regard to several different performance characteristics.

Throughout the course of the project, we have provided the following contributions:

- Fully working Python code implementing every major algorithm in the Private Stream Aggregation protocol. To the best of our knowledge this is the first full implementation of these algorithms.
- Helper code to assist in the setup and generation of shares for the RASS protocol.
- A basic simulation benchmark covering the communication characteristics of the RASS protocol.
- Useful and accurate benchmarks covering many aspects of the PSA protocol and its computational complexity.
- Data and comparisons of the algorithm’s performances with regard to changes in various key parameters.
- A direct comparison of each algorithm’s performance vs. the other.
- The code necessary to generate further benchmark data if the data provided does not satisfy.
- Thorough but simple explanations of the operation of each protocol along with step by step examples.

The primary difficulties encountered during this project were mostly issues relating to the correct and accurate implementation of the different PSA protocols. Being a rather complex protocol, there are many small things one might overlook in their implementation that turn out to be absolutely crucial for an accurate and correct solution. There were many occasions when a feature was thought to be complete and functional, only to turn out that some seemingly obscure mathematical principal or quirk was invalidated by the implementation, causing everything to fail.

The biggest barrier to success that was encountered was undoubtedly the work involved in order to implement a custom communication protocol using VIFF. Such a thing is not documented anywhere, nor even suggested, and as a result a full implementation including communication was unachievable in the given time constraints of the project. It is definitely possible to do, however, especially if one were to be familiar with the Twisted networking framework that VIFF uses to drive its out of the box communications. This would be the biggest unit of potential future work, and would provide a much more accurate series of benchmarks for RASS in particular.

There is also work that could be done in further improving the performance of the PSA algorithms, specifically the AggrDec decryption algorithm. Though having adapted an implementation of Pollard’s lambda algorithm has provided an immense speedup, there
exist other methods of calculating discrete logarithms which may provide superior performance. Pollard’s lambda algorithm also has the potential to be parallelized.

A further benchmark that wasn’t implemented due to time constraints was that of memory usage by the protocols. This is one which could provide very useful data too.

Running the benchmarks on various machines of different specifications would be a valuable exercise. The machine used for the benchmarks during this project was one with relatively high computation power, and it would be worthwhile to see what results were obtained for the various different benchmarks on machines with differing abilities in this regard.

Of course, one could also look to implement and benchmark other secure summation protocols besides PSA and RASS, to provide a more thorough set of comparisons and data on when one algorithm would be more suited for use than another.
References