Crux: Privacy-preserving Statistics for TOR

MSc in Information Security

Computer Science Department

University College London

Vasileios Mavroudis

Supervisor: George Danezis

Abstract

This work introduces Crux, a scalable and modular system which computes statistics on the activity of hidden services operating in Tor anonymity network.

Until now, the Tor network administrators collected only generic activity data, which do not pose a threat to the users' privacy. However, recently a need for more detailed monitoring and statistics emerged. The collection of these data by the relays is trivial, but the aggregation and the computation of statistics from them can potentially de-anonymize individual Tor users or hidden services. For this reason, all computations need to be performed in a manner which preserves the users' privacy but also gives accurate results.

Crux offers a solution to this problem by employing techniques which enable the aggregation of encrypted measurements and the computation of various statistics (e.g. median, variance) while preserving the users' privacy. To achieve this, it utilises an additively homomorphic scheme and a number of data structures depending on the data type required for the statistic. In this work, we present a complete implementation of Crux and demonstrate its performance both in terms of efficiency and in terms of security.

Keywords: TOR, privacy-preserving computations, statistics, secure multi-party computations, homomorphic encryption, stream processing algorithms

Acknowledgement

First and foremost, I would like to thank my supervisor, Dr. George Danezis, for his guidance and patience. Our meetings have been an inspiration for me and I was always looking forward to them. Dr. Danezis helped me improve my research skills and stayed on top of my work, while giving me the freedom to take the important design decisions myself.

I should also mention the support and assistance I received from both the academic and administrative stuff of the computer science department and the Bloomsbury campus. This year wouldn't have been the same without you.

Last but not the least, I am really indebted to my family: Konstantinos, Anastasia, Lara and Natasa, for their endless support, encouragement and motivation throughout all these years. Thank you.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and Goal

During the last decade the need for anonymous communication and privacy protection became more prevalent than ever. Tor (i.e. The Onion Router) [36] is an anonymity network which was first introduced in 2002 and intends to offer a solution to the above problem. In the last few years, Tor increased its user base rapidly and currently it consists of more than six thousand relays [1] which route the users' traffic in order to conceal its activity.

However, the administration of such a large network which routes sensitive information is a complex and tedious procedure. To assist this effort and to enable the monitoring of the hidden services, the Tor relays are currently collecting some generic statistics on the activity of the network. These measurements are carefully selected so that they do not pose a threat to the users anonymity or the hidden services identity, but for the same reason they provide only a high level overview of the network activity.

Lately an effort is being made to improve the monitoring of the Tor hidden services [52]. The role of Tor hidden services is to provide network services while protecting the network server's identity [52]. More specifically, the measurements will be extended to include the load and the usage of the services. However, these statistics could potentially reveal sensitive information about the users and/or the network resources and hence the aggregation and the processing should be performed in a privacy-preserving manner.

In this work, we initially review the relevant literature and related projects and systems which were applied in similar cases (a more in depth review of the field can be also found in our previous work [62]). Subsequently, we examine the needs and the privacy requirements of Tor and based on them we design a scheme aiming to provide a solution to the problem of privacy-preserving processing of the measurements. We then implement Crux, which is a system based on our design and finally go through a detailed evaluation where we examine its performance and identify and fix any shortcomings found.

Crux is a privacy preserving system which is able to compute a variety of statistics by aggregating and processing the activity measurements made by multiple Tor relays. Its application would improve the hidden service monitoring capabilities, enable optimizations by the developers, help identify bugs, and potentially uncover ongoing attacks.

1.2 Structure

This work consists of five main chapters. In chapter 2, we introduce some fundamental concepts, outline various techniques for privacy-preserving computations and statistics along with their practical applications and shortcomings and examine the most commonly used stream processing algorithm. Moreover, chapter 2 outlines the way Tor works and its security assumptions. Chapter 3, contains the requirements and the goals that the proposed system should satisfy. In Chapter 4, we introduce the system design, outline its components and communication channels and present its key points, while chapter 5 contains the details of the implemetation. Finally, in Chapter 6 we perform an evaluation of our system, suggest improvements, and propose future directions based on our findings and the recent developments in the field.

2 Background

In this section, at first we introduce some definitions which will be used in the rest of this work. Subsequently, we review privacy-preserving schemes and query anonymization techniques and examine their potential applications and their limitations. Finally, we present some stream processing algorithms, privacy processing statistics, while we also review the Tor anonymity network.

2.1 Definitions

Warren in his work "The Right To Privacy" provided one of the first formal realisations of privacy. In particular, he defines privacy as the "right to be let alone" held by every individual. His definition was inspired by the technological advancements of the era (e.g.

photography) which were used by journalists in an immoral way. This first definition is not very accurate nowadays, but more recent works attempted more up to date descriptions. For instance the authors in [69] define privacy as:

> "The right to privacy is our right to keep a domain around us, which includes all those things that are part of us, such as our body, home, property, thoughts, feelings, secrets and identity. The right to privacy gives us the ability to choose which parts in this domain can be accessed by others, and to control the extent, manner and timing of the use of those parts we choose to disclose."[69]

This definition is also compatible with the problems faced by the organisations which want to publicly release datasets or outsource the computations on sensitive data. According to the definition, since the datasets contain the personal data of individuals, the release authority must ensure that their right to privacy is not violated.

Another term which we will use throughout this work [62] is *data subject*, which refers to:

> "An identified natural person or a natural person who can be identified, directly or indirectly, by means reasonably likely to be used by the controller or by any other natural or legal person, in particular by reference to an identification number, location data, online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that person."[24]

2.2 Privacy-preserving Computations

The field of privacy-preserving computations started in 1982 when the millionaire problem was introduced by Yao [78]. The setting of this problem is the following: Given two parties A and B which possess two values a and b respectively, they want to determine if $a \leq b$ or not. The limitation is that each of them wants to keep his value secret.

A straightforward solution would involve a trusted third party which compares the

values and announces the result. However, the assumption of the problem is that there is not such trusted authority, and this makes the problem non-trivial. Solutions to this class of problems are provided by the secure multi-party computations (SMPC) subfield of cryptography. In particular, SMPC enable multiple actors to jointly compute a function over the secrets each one of them holds, without revealing any of them. Such algorithms are applicable in privacy-preserving computations, since they enable the processing of data while preserving the of the data subjects.

Privacy-preserving schemes fall under three main categories: homomorphic schemes, secret-sharing schemes and circuit-based ones and all of them consider one of the two adversarial models:

- *Honest but curious (HBC)*: one or more parties execute the protocols properly but trie to observe and collect as much information as possible
- *Malicious*: one or more parties are actively malicious meaning that they may drop, alter or send malicious messages to extract as much information as possible.

In the following sections we present the most well-researched types of SMPC schemes, and some of the most important works in each of them. [62]

2.2.1 Homomorphic Encryption

The first homomorphic cryptographic scheme was proposed by Rivest et al. in [73] and was targeted towards searches on encrypted data. However, as the properties of such schemes were studied more closely, later works used them to perform require privacy-preserving computations (e.g. voting protocols). Homomorphic schemes provide a way for multiple parties to compute one function on their data, while maintaining their secrecy. More formally, let k secret values $x_1, .., x_k$, one from each distinct party and a function f where $y = f(x_1, .., x_k)$. Based on these, we need to define a privacy-preserving protocol P which accurately computes the value y and doesn't reveal any information on the secrets. We should also note that the execution traces of the protocol are also considered an information source for the adversary. There are two variations of the above definition: one where the parties trust each other to execute the protocol correctly and one where the parties

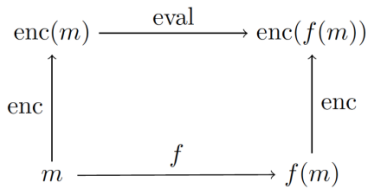

Figure 1: Demonstration of the *eval* function.

are actively malicious and may divert from proper execution.

A homomorphic scheme comprises of the following algorithms: *KeyGen*, *Enc*, *Dec* and *Eval* [73] [43] [18]. The first three algorithms are commonly seen in classic cryptographic schemes, however, the evaluation algorithm *Eval* is not. It takes as input a public key p, multiple ciphertexts $c_1, .., c_k$ and a circuit C, and produces an output c, where $c \leftarrow Eval(C, p, c_1, ..c_k)$. The validity of the scheme, for a circuit C, a key pair (*pub*,*priv*), and plaintexts $m_1, .., m_k$ is verified if given $c_i \leftarrow Enc_p(m_i)$ for $i = 1..n$ and $c \leftarrow Eval(C, p, c_1, .., c_n)$, it holds that $Dec_s(c) = C(m_1, .., m_n)$ (figure 1).

Moreover, such schemes are categorized by the types of operations which can be performed on and between the ciphertexts they produce. In particular, they are called *additive* for addition, *multiplicative* for multiplication and *fully-homomorphic* if they support both operations. Initially homomorphic schemes were partially-homomorphic meaning they only supported addition or multiplication and fully homomorphic schemes were introduced much later.

A popular work in the area is *Helios* [2]. Helios is a voting system, which stores ballots publicly and enables everyone to verify the computations made. As such, it makes use of the properties of a homomorphic encryption scheme but it is not considered a pure multi-party computation system. The homomorphic scheme used is additive and also offers some additional features which make it capable of supporting online elections. More specifically, Helios enables the public to retrieve the ciphertext of the votes, and verify the final result of the elections and that his/her vote was considered during the computation

of the final result. Since the introduction of the system various works were published to improve or attack Helios [11] [53].

Even though Helios was very successful most applications require both addition and multiplication and hence, partially-homomorphic schemes are often too restrictive as they cannot express complex equations and functions. *Fully-homomorphic* schemes solve this problem, as they support both addition and multiplication of the ciphertexts. However, unlike partially homomorphic schemes, fully-homomorphic schemes took much longer to be introduced and there is a thirty year gap between the introduction of the two types of schemes.

The Boneh-Goh-Nissim cryptosystem [16] was one of the very first fully-homomorphic schemes, and it supports both addition and multiplication. Unfortunately, it has severe limitations, as it can only perform one multiplication but unlimited amount of additions. Furthermore, its message space is limited as the discrete logarithm of the ciphertext has to be computed during the decryption, but this characteristic is present in newer schemes too. Gentry in [47] [48] was the first to introduce a scheme supporting unlimited number of additions and multiplications between the ciphertexts. His contribution was considered a milestone for the field as until then it was debatable if a fully operational scheme was possible. The scheme is based on problems from the field of bootstrappable encryption and ideal lattices. However, it was also very complicated and inefficient to be practically applicable. After this first scheme, more complete solutions were proposed by other researchers. For example, Brenner et al. in [17] designed a method to execute a program using as input sensitive data while protecting the confidentiality of both the data and the program. Their implementation, provided a runtime environment for the execution of the encrypted programs and the assembler for the generation of the program's machine code. The significant contribution of this work was that the encrypted program supported dynamic parameters, branchings at runtime and even self-modifying code, while previous solutions were limited to one-time circuits.

Currently, many researchers work to improve the efficiency of the already existing fully-homomorphic schemes as even though many of them are provably secure and very flexible, their use in practical applications (e.g. for privacy-preserving computations) is limited due to their low efficiency [68].One influential work in this direction is the SPDZ protocol [31]. SPDZ is a multiparty computation protocol which is secure against both honest but curious adversaries and actively malicious ones, as long as one of the parties remains honest. The protocol utilizes both homomorphic encryption and secret sharing (see subsection 2.2.2) techniques to provide a complete . SPDZ comprises of two phases: a preprocessing one and an online one. The online phase is one of the main contributions of the authors as they managed to reduce its runtime significantly. Additionally, they suggest that the offline phase is the proper way to use fully homomorphic encryption schemes (see also below) in practice, since many optimizations can be performed on the circuit evaluation process (e.g. parallelization). Later Damgaard [30] et al. further improved the protocol by enabling the reuse of the preprocessed data for the evaluation of more than one functions on the online phase.

2.2.2 Secret Sharing Schemes

Secret sharing schemes provide a way to share a secret s between a number of parties, in such a manner that the secret can be recovered only if all parties combine their shares. In some settings, it may be required that at least k parties (where $k \leq n$) to combine their shares to recover the secret. In any case, if less than k parties to combine their shares the recovery fails and no information about the secret is leaked.

Shamir in [74] and Blakley in [12] introduced the first secret sharing schemes simultaneously. However, their schemes followed slightly different designs to achieve the same result. Shamir exploited the fact that k points can define a polynomial of degree $k-1$. He then built his scheme onto the assumption that for a secret recoverable by at least k parties, one can define a polynomial $f(x) = s + r_0 x + ... + r_{k-1} x^{k-1}$, where r_i denote some randomly chosen values and the secret s is fixed as the constant part in the polynomial.

On the other hand, Blakley's encoded the secret as a coordinate of the intersection point of k hyperplanes with $k-1$ dimensions [12].

Subsequently, other researchers introduced more complex schemes offering additional features. A percentage of them enable the users to even perform numerical operations on the secret shares, which is very helpful feature for privacy-preserving computation applications. One such work is "Sharemind" [13]. Sharemind is a framework capable of collecting sensitive data and then processing them in a privacy-preserving manner. First the sensitive data are "split" between multiple entities called miners. Then a secure multi-party computation protocol is executed on the shares and the final result is derived. These protocols are developed using the programming library provided by Sharemind and the resulting programs are executed in a specialized runtime environment. Sharemind was designed with scalability in mind and supports large-scale share computations. Moreover, it features a full set of share operations including share addition and multiplication between shares and with a constant. Even though, the framework appears to be very flexible in terms of operations it suffers from two limitations. The fact that is supports at most 3 computing parties and that is considers only semi-honest adversaries.

Additionally, the authors of [15] and [14], construct a privacy-preserving auction system, and deploy the their implementation to solve a real-life problem in Denmark. More specifically, they built an electronic double-auction platform using a three-party solution along with secret sharing mechanisms. Their implementation uses a (non-interactive) verifiable secret sharing schemem, which builds upon the scheme of Shamir and supports addition, multiplication and even comparison between the shares.

Moreover, in 2013 Danezis et. al [34] applied a secret-sharing protocol to process electricity meter readings in a privacy-preserving manner. According to their description, smart meters are both a useful source of data and a potential threat of the users' privacy. This is because their advanced monitoring capabilities (compared to simple meters) enables them to provide detailed overview of electricity consumption during the day. The authors

suggest a protocol with the following components: the smart meters, multiple trusted authorities, a storage server and some auxiliary services. Additionally, the authors assume that all the components are honest-but-curious, when executing the operations of the additive secret sharing scheme they use.

The role of the meters is quite simple as they just send periodically shares of their measurements to the authorities. Once a user sends a statistic request to the storage service, the service notifies the authorities and they do the necessary operations on the shares they hold. Then the service collects the shares, combines them, and returns the result. Apart from the immediate practical application of their design the novel feature the authors introduce is that their scheme supports a very wide range of statistics (e.g. linear and non-linear functions, aggregates, boolean circuits) to be computed.

Finally, in the last few years, researchers also attempt to combine secret sharing with other techniques in order to develop more efficient and secure multi-party computation schemes. On such scheme was suggested by Keller et. al in [57]. The core of their design comprises of the compiler and the runtime execution environment which executes the protocols using preprocessing models to decrease the runtime and unlike "Sharemind" works under the assumption of actively malicious adversaries. [62]

2.2.3 Garbled Circuits

The third category of schemes is garbled circuits, which were first introduced by Yao in [78]. The great majority of garbled circuit schemes solves problems involving only two-party computations. This makes them less usable for privacy-preserving computation scenarios were the data are held by more than two parties. Each garbled circuit involves two actors A and B who desire to jointly compute a function f. The function will use as an input their secret values a and b, but, similarly with the millionaire problem, they do not want to reveal their values.

During the execution of a garbled circuit protocol, A prepares a (boolean) circuit of the function f, which accepts two inputs a and b, and produces multiple ciphertexts from

it. For instance, an "AND" gate which gets encrypted produces 4 ciphertexts as such: $c_{ij} = Enc_{X_i,Y_j}(Z_k)\ \forall i,j = \{0,1\}$ and $k = i \wedge j$ [70] [62]. These 4 ciphertexts are considered the garbled circuit which is then sent to B. In addition to this, B receives a key X_a which is paired with the input of A to the circuit. Once B has received the circuit and A's key, he needs to obtain his own key Y_b. To do this he initiates an oblivious transfer with A. Upon the reception of Y_b, B promptly attempts to decrypt all four ciphertexts. This procedure will result in only one successful decryption and this will be the output of f when applied on the inputs a and b. Garbled circuit usually require $4k$ encryptions for a circuit comprising of k gates.

As said, only a few works use garbled circuits for privacy-preserving computations and involve more than two parties. One of them is FairplayMP [9] which extends the two-party computation system Fairplay [9] and is secure against semi-honest adversaries. [62]

2.3 Differential privacy

As discussed above, the goal of releasing a dataset which both contains useful information and preserves the privacy of the data subjects is hard to achieve. For this reason, an alternative method of sharing a dataset was proposed. More specifically, instead of publishing the whole dataset, the researchers are able submit queries to a server which processes the dataset and returns the processing result. A naive implementation would prohibit queries on individual subject and would force the users to submit queries involving at least k data subjects (e.g. "what is the mean electricity consumption for n houses in the street?"). However, as proved by the authors of [35] these limitation do not provide enough protection to the data subjects and to make matters worse potential breaches cannot be efficiently detected.

A more advanced approach is *differential privacy* (DP) [38] [37] which allows the users to query the database while it provides at least a minimum degree of privacy protection for the subjects. Similarly the previous techniques, differential privacy treats the dataset as a table where each individual is included in one row.

The guarantee of DP is that given two datasets D_1 and D_2, where $D_2 = D_1 \cup \{d\}$

where d is an entry for a single individual (i.e. an element) and a function $K = f(D)$, K provides ϵ-differential privacy if for any D_1, D_2 it holds that [62]:

$$\frac{Pr[K|D_1]}{Pr[K|D_2]} \leq e^{\epsilon} \tag{1}$$

where $e^{\epsilon} \approx 1 + \epsilon$ for small value of ϵ.

In other words, a differential privacy mechanism guarantees that the result of K applied on a dataset which include the element d (D_2), and the result of K when applied on the same dataset without the element d (D_1), will at most differ by ϵ. ϵ is also called the sensitivity of K and depends on the maximum difference a single record could make on the output of K. In practice, DP mechanisms utilise a variety of techniques to perturb the output of the query (e.g. Sample and Aggregate, Laplace Noise) based on the value of ϵ required.

It is important to note that differential privacy can also be bypassed by an adversary who submits multiple queries until the information leaked is non-negligible. A mitigation of this attack is offered by the *privacy budget*. The privacy budget treats the queries as a finite resource and sets an upper bound to the number of queries a user can submit. Of course, this also limits usability offered to honest users and introduces a tradeoff between utility and data privacy [63] [45].

2.4 Stream Processing Algorithms

Even though non-trivial streaming algorithms were first introduced in 1970, it wasn't until 1996 that their applications in big data were discovered [5] [72]. Streaming algorithms are concerned with the computation of statistical operations on the elements of a data stream. However, there are specific constraints on their operation, which make their design and development interesting and non-trivial. More specifically, streaming algorithms operate by sequentially analysing the items in a data stream and in most use cases, examine each item of the stream only once. Additionally, the available memory space is sub-linear to the space needed to store the whole data sequence, meaning that it cannot simply store the whole sequence but instead it should in run a function on each element and either store the

whole element (sampling) or extract only the information needed (sketches). Additionally, there are often some limitations on the processing runtime which can be allocated for each element of the data stream. [3]

As a result of these constraints, stream algorithms often provide only an approximation of the correct answer and hence the discrepancy from the correct result is one of the criteria used to determine the performance of the algorithm. Furthermore, when developing stream processing algorithms three different data streaming models are considered [67]:

- The Time-Series Model where the elements of D arrive sorted by i
- The Cash Register Model where the elements of D arrive in an arbitrary order
- The Turnstile Model where elements appearing in the stream may be removed later (i.e. updates)

2.4.1 Sampling

Sampling is one of the techniques which were traditionally used in problems with data streams. The main concept of sampling is that given a dataset we collect a random sample S with size m in an attempt to capture specific attributes of the entire set. If the sample is indeed representative of the data stream then it can be used to answer queries regarding the stream.

However, in most practical applications the size of the stream is either not known in advance or the stream provides new data constantly (e.g. in the core of telecommunication networks). This makes random sampling less straightforward, as we need to maintain a sample of a fixed size over a data stream of unknown size. In such cases, the most commonly used technique is *reservoir sampling*, which was introduced by Vitter in [75]. In reservoir sampling given a data stream D of n items, we generate a sample S of size m ($m << n$) by constantly collecting items observed in the stream. In other words, the l-th item of the stream has probability to m/l to end up in the sample S. The limitations of reservoir sampling are that it does not account for data deletions and duplicate entries

may waste part of the limited memory space.

To address these shortcomings other sampling techniques have been proposed such as concise and counting samples [50], moving window sampling [8], Wavelet-Based Histograms [61] and approximate histograms [51]. However, despite the fact that sampling is a well-researched area, it is very inefficient when applied in specific classes of problems as it requires nearly linear amount of memory.

2.4.2 Sketches

In the past few years, much research has been done on streaming summaries which use sketches. Similarly with sampling, sketches are suitable for the data streaming scenario, as they are able to extract useful information from large quantities of data and subsequently answer queries, using a sub-linear amount of memory. More specifically, sketch-based algorithms aim to create a synopsis/summary of the observed data, that fits in the limited amount of memory which is available to the algorithm [25]. On each new item of the stream that is observed the sketch is updated, so that is represents all items seen so far. At any point, the algorithm can use the current sketch to answer queries by applying functions on the sketch. From the above, it becomes apparent that modern sketching techniques, focus mainly on the turnstile model where the elements can be updates with either solely non-negative i values (strict model) or without this restriction (general model).

Sketches were first introduced in the 1980 and they gradually gained popularity through the years. One of the first sketches introduced is the Flajolet-Martin sketch [42] [41], which approximates the distinct count of the data stream elements using a hash-based probabilistic counter. Furthermore, another research direction was proposed by Whang et al. [77] early on. His method, *linear counting* uses a bloom filter with $k = 1$ to achieve compact set representation. One of the characteristics of this class of methods is that they require space linear to the cardinality of the attribute examined [26]. This comes also with the limitation that the attribute cardinality should be known from before, so that the filter will have an appropriate size. Apart from the above techniques numerous approaches and their

variations emerged through the years (e.g. AMS Sketches [4] and the improved version [26]) each of which is efficient in a different stream model and for different types of queries.

In this work, we mainly focus on *linear sketches* which can be thought as the result of a linear transformation applied on the input data stream. In other words, to generate the sketch of a data stream, we need to model a relation as a vector and then multiply it with the data represented as a matrix (figure 2) [25]. An advantage of linear sketches is that as new data are added by the stream the sketch can be updated without recomputing the transformation for all the data. Instead the transformation is applied to the change in the data and the result is added to the sketch. Additionally, the sum of two sketches originating from two district data streams, corresponds to the union of these two streams.[60]

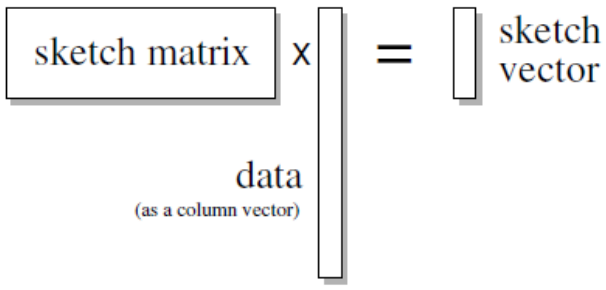

Figure 2: Overview of linear sketching [25]

A variation of linear sketches is of special interest for our purposes as it estimates frequency moments of the input data stream and hence can be used to approximate various statistics on the data (e.g., percentiles, median). More formally, the two basic components of a sketch are the data stream D containing a set of points U where $U = \{1, 2, ..., M\}$, and a function $f(\cdot)$, which estimates the number of elements in D which hold a specific value from U. Based on this model a number of sketching techniques can be used. Here we present *count-min sketch* [28] and *count sketch*.

Count-Min Sketch is the newest of the two and works by maintaining an array C of counters (i.e. the sketch) with size $d \times w$, where d is the number of rows and w is the number

of columns. A hash function $h(\cdot)$ is attributed to each row and thus d hash functions are used. We denote the hash function of the j-th row as $h_j(\cdot)$. Each hash function maps its input domain U within the range $\{1, .., w\}$. To build the array C equation 2 is used [27]:

$$C[j,k] \ = \sum_{1 \leq i \leq M : h_j(i) = k} f(i) \tag{2}$$

which in other words means that the l-th column in a row contains the number of elements in the stream, which the hash function, corresponding to this specific row, maps to l. The update of C as new elements appear in the stream is straightforward and is depicted in figure 3.

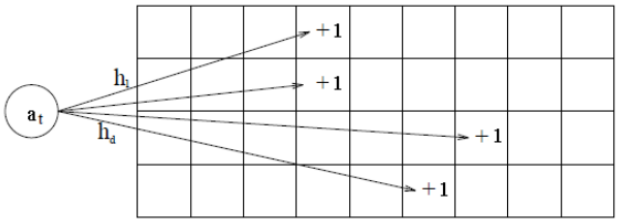

Figure 3: Insert operation in count-min sketch [25]

Moreover, the estimation of an $f(i)$ is requires building a set of values $\{C[1, h_1(i)],\ C[2, h_2(i)],\ \ldots, C[d, h_d(i)]$ and taking the smallest element of that set.

Count sketch [21] works exactly as count-min sketch, except from the estimation procedure. In count-sketch the estimation for a row j is computed as such:

Given $h_j(i)$ and $h'_j(i)$, where

$$h'_j(i) = \begin{cases} h_j(i) - 1, & if\ h_j(i)\ mod\ 2 = 0 \\ h_j(i) + 1, & if\ h_j(i)\ mod\ 2 = 1 \end{cases} \tag{3}$$

the estimate for the row is $C[j, h_j(i)] - C[j, h'_j(i)]$ [27]. This approach aims to increase the accuracy by mitigating the noise that is introduced from items which collide with i.

2.5 Privacy Preserving Statistics

In the previous sections we have studied various schemes and discussed their potential applications in specific real life problems. For instance, we examined homomorphic schemes with applications in voting [2], secret sharing techniques used in smart-metering [34] and multiparty computation schemes used to coordinate auctions [15]. In this section we focus on privacy preserving statistics (PPS), which aim to provide ways to estimate or approximate statistical indices in a variety of real-life scenarios. We should note that PPS are usually designed with a specific application in mind (e.g. smart-meters) but they are often applicable in different usage scenarios.

The majority of the works on privacy-preserving collection of statistics uses either homomorphic encryption or secret sharing schemes to realise the security properties needed in such applications. More specifically, such designs feature an aggregator or coordinating entity, which is usually assumed to be semi-honest, and use it to sum the encrypted readings from multiple users. One of the areas for which many PPS schemes have been proposed is smart-metering and power consumption. In such a work, Kursawe et. al [58] introduced a set of protocols, which aggregate meter measurements and enable the detection of fraud as well as the computation of various statistics from the smart-meter measurements. Moreover, Erkin and Tsudik [39] take a slightly different approach and propose techniques which are based on peer to peer interaction of the meters and eliminates the need for on-line aggregators. The authors in [59] propose a solution to the problem of personal information leak which occurs when extracting statistical information from smart-meter data. More specifically, they introduce a method which uses order preserving encryption and a delta encoding scheme to realize the operation of identifying the maximum consumption of a smart meter. The paper argues that this operation is of interest for the energy providers and that the scheme effectively protects the privacy of the data subjects and does not leak any information.

Apart from smart-metering, privacy preserving statistics have various other application areas. For instance, the authors in [19] use a simple privacy-preserving scheme to address the problem of data aggregation in wireless sensor networks. Their scheme is able to

operate on low bandwidth which is an important requirement for such applications and is able to compute a number of statistics on the measurements such as the variance, the standard deviation and the mean.

There are also works which aim to improve other aspects of the existing schemes. In [56], Jawurek et al. introduce a realtime statistics scheme which builds upon already existing methods. First, the authors examine the state of the art and determine that very few schemes account for errors during protocol execution, and those that do suffer from substantial performance decrease. Based on these observations, they developed a system which solves the aforementioned problems, simplifies the key-management procedures by using a key authority and provides a variety of statistical functions. Finally, the authors claim that their scheme offers the very interesting feature of unbounded number of statistics calculations, even though they utilise differential privacy techniques during the calculations.

In addition to homomorphism and secret sharing, differential privacy techniques have also been used to add noise to the dataset. In most schemes, the noise is added by the aggregator. One representative work is RAPPOR [40], which is a framework providing differentially-private methods for the collection of statistics based on input perturbation. RAPPOR allows the study of client data as a whole, but it does not permit the examination of the measurements from individual clients. Additionally, Cormode et. al [29] provide a number of methods which aim to address the problem of applying differentially-private techniques in a sparse dataset, without vastly increasing the size of the released dataset.

It should be noted that differential privacy does not provide any protection to the measurements by the data sources and hence it should by always combined with other techniques, if that is a requirement. An example of such a combination is the work from Chan et al. [20] which combines differential privacy techniques with applied cryptography and proposes privacy preserving mechanisms applicable in a variety of fields (e.g. market research, energy grid).

2.6 The Onion Router - Tor

Tor [36] is a network of relays which handles internet traffic and aims to conceal the location of the user and the transmitted data from specific types of attackers. Anonymity systems, such as Tor were first introduced by Chaum with the *MixNet* scheme [23]. Such systems aim to hide the sender and recipient connection and the exchanged data by encapsulating the packets in an onion-like manner using relays and public key cryptography. From this initial design two approaches emerged:

1. High latency systems such as Mixminion [33] and Mixmaster [66], which offer increased anonymity and are able to withstand attacks even from a global attacker but suffer from limited practical utility due to their interaction lag.
2. Low latency systems (such as Tor [36]) which offer increased usability, at the cost of a weaker security assumption (i.e. they are not secure against global adversaries).

In the Tor network, the relays are called *Onion Routers (OR)*. The client selects the OR path it is going to use, and subsequently builds a circuit in which each node knows only its predecessor and successor. The circuit offers bidirectional communication by relaying packets between the entry and the exit node. To ensure the privacy of the communication the packets are encrypted by each OR, reordered and possibly (slightly) delayed before they are transmitted to the next node [36].

As a result, each relay participating in the Tor network is able to extract some statistical information from the traffic it routes on the load and the usage of hidden services. Even though this information is useful for the network operators, it can also be used to breach the anonymity of individual users and services. This work is based on this realization, and introduces a processing scheme which is able to use these data to extract information about the network activity without jeopardizing the privacy of the users.

3 Requirements & Design Goals

This chapter formally outlines the needs that our system aims to cover and the assumptions it operates on. Additionally, we also detail the different types of requirements which should be covered by the final implementation (see also chapter 6). On the second half of this chapter, we also discuss the threat model and compare it with the threat model of Tor.

3.1 Goals

Crux aims to provide a way for the Tor administrators to compute useful statistics for the hidden services based on information collected by the network relays, without affecting the level of security provided by Tor. Apart from this main goal, a number of considerations of various types, directed the final design of the proposed system.

3.2 Requirements

In order to develop a simple and stable system capable of operate on top of a large and peculiar, in its nature, network, we set a number of requirements which specify all aspects of the design and the implementation.

To begin with, in terms of *applicability*, the system should be easy to deploy on an already operating relay. This implies that it will not require any special hardware to run and the computational requirements will be low. Additionally, since the network bandwidth is often limited and consumed mostly by Tor related traffic the bandwidth usage should be as low as possible. Furthermore, low maintenance requirements and ease of deployment will positively impact the number of node administrators which decide to install this additional package on their node. Moreover, all the components should be robust and especially the parts which are deployed on the Tor relays. This is because any crashes, memory leaks etc will affect not only the operation of the statistics computation system but also the performance of the relay as part of Tor. The above requirements are important because the nodes are maintained by volunteers and hence most nodes run on low-end hardware with low computational capacity.

In addition to all the above, the design should be as simple as possible. In particular, all the security parameters and protocols used in subsystems should be well-documented and as simple as possible. Another design requirement is modularity. Instead of building large monolithic blocks, we aim to distribute the functionality between and within the components in such a way that will enhance the robustness of the system and will facilitate future improvements. Another reason, simplicity and modularity are important is the fact that the implemented system will be open source. A modular, well-documented and not unnecessarily complex project is more likely to receive contributions both in the form of new features and bug fixes.

Another requirement concerns usability. The end user and the system administrators should be able to submit queries without having any knowledge of the underlying cryptographic schemes or understanding of the operation of the whole system. For this reason, it is important that the user module supports a variety of operating systems and does not require complex installation procedures (e.g. kernel patches). Of course, it is understandable that a different client will be used for each operating system but it is important to make sure that no (or at least very few) OS-specific characteristics/functionality are used in the design.

Finally, the security requirements are very similar with those of Tor[36]. More specifically, the assumptions about the adversaries should be at least as weak as those found in the threat model of Tor. In the following section we outline our threat model and discuss how it relates with the model presented in the Tor paper [36].

3.3 Security Policy & Threat model

As pointed out in section 3.2 the threat model of our system is very similar to the one assumed by Tor. This is due to the fact that if the security assumptions of Crux were stronger than those considered by Tor then our system could become a weak point which an adversary could exploit to attack Tor.

To begin with, the system involves only a small number of principals. In particular, the entities who interact directly or indirectly with it are:

1. the user who submits queries for statistics. In practice, he would likely be a network administrator or a hidden service maintainer but since the computation results are public it can be anyone with an interest in the area (e.g. a researcher).
2. the administrators of the different components. This includes all the volunteers who run Tor nodes, which support Crux and those who operate individual nodes such as the query processors and key authorities (see also section 4.1).
3. the Tor user. Even though the Tor user does not interact with Crux directly, his/her actions affect the final result of the computations.

The assets which we want to protect from adversaries are: The confidentiality and the integrity of the Tor network traffic, the users' data and its connection details (e.g. the circuit and the nodes used), the relays and hidden services and their details (e.g. network activity, location), and the reliable operation of the whole network. Our main goal is to provide an additional service to the Tor users and administrators, however, this should not affect the operation or the security of Tor in any way.

For our threat model, we assume an adversary who is able to observe all the traffic between the components of our system and has perfect knowledge of the topology and the role of each component. The attacker is not able to modify or delete traffic. However, he may gain the control of either a network node or of one or more key authorities, with one important restriction. We assume that an adversary can gain control of all but one key authorities at most, and that at least two Tor relays participating in the computation remain out of his control. This is in accordance with the threat model of Tor were the adversary is assumed to be unable to observe the whole network, and while he may operate a small number of malicious nodes he cannot control the majority of the network relays [36]. Moreover, the adversary may be able to observe the traffic that goes through the query processor and the computations it performs and have full knowledge of the computation protocols.

We should also note that Crux is a system for the computation of various statistics based on the users traffic which also preserves the user privacy. Hence, the system is

designed with this goal in mind and it cannot prevent the administrators from computing statistics which violate the security properties of Tor.

In section 6, we evaluate the performance of our implementation with regards to the threat model presented above.

4 System Design

This chapter presents our proposed scheme, discusses various design details, and presents the details of the components and the communication channels.

4.1 Components

Our design comprises of three main components which realize different operations and hold a different number of security properties. Figure 4 provides an overview of the system design and the component interactions. As seen, the interaction between the components is limited only to what is needed for the computation of the final result and no other communication between the components is allowed. In practice, some of these components can be merged without violating the security properties of the system.

4.1.1 Tor Relay

The main functionality of Tor relays is to route encrypted user traffic in the Tor network, while at the same time protecting the identity of the hidden services. Our system uses the traffic statistics collected by these relays. It's main operation includes accepting service requests, fetching of the statistics requested, encrypting them and returning the ciphertext to the component initiating the request. To achieve this functionality, each relay has to support the following functions:

- Fetch the traffic statistics collected/generated by the Tor software, without access to any other traffic data or information. These data can be either in the form of single values or in more complex structures such as sketches.

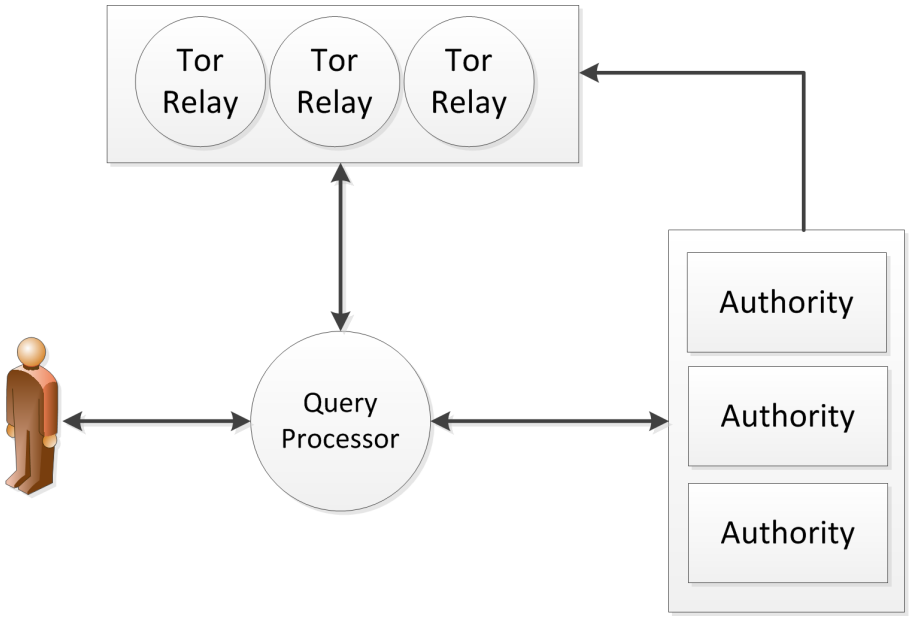

Figure 4: Map of the possible interactions between the components.

- Collect public keys from trusted key authorities. This is done by sending a request to each authority and receiving a response with a (potentially ephemeral) public key, signed using the certificate from the trusted authority. If signed public keys are used, then the relay should also be able to verify the validity of the signature.
- Use the individual public keys collected to generate a common public encryption key, which will be used for any encryption operations.
- Encrypt any value or sketch using the common encryption key, and generate a randomized ciphertext.
- Accept service requests from the query processor (section 4.1.3), process them, and respond with the corresponding ciphertext.

The above functionality is not supported by the Tor software ran by the relays and hence additional software has to be installed (see also section 5).

According to the threat model every deployment must include at least two relays. Additionally, our security assumption for the relays is that an attacker cannot control more than $N - 2$, given that the number of relays is N. Note that this assumption is weaker than the one made in the Tor threat model. In such a case, the attacker can observe the

encrypted traffic going through the relays he controls but he cannot exploit our system to infer the traffic measurements from other individual relays. In the context of our design, only the query processor can initiate interaction with the relays and send service requests to them.

4.1.2 Key Authority

The key authority operates as a separate component which realises the following functionality:

- Generate a (potentially ephemeral) pair of public and private key and maintain a signed version of the public key. More specifically, the public key is signed using a certificate issued by a trusted authority.
- Receive a ciphertext and decrypt it using the private key from the (ephemeral) key pair. If the key pair is ephemeral, then allow for a pre-determined number of decryptions before a new pair is generated.
- Upon request the authority sends its public key to the component, which initiated the request.

We consider the key authorities to be honest but curious. This means that they execute the above functions as expected but at the same time they try to extract as much information as possible by observing the execution traces. Additionally, they are non-colluding meaning that they do not cooperate in any way to combine the information observed. The key authorities are accessible only by the relays running the Crux software and the query processor presented below.

4.1.3 Query Processor

The query processor is the core component of our system as it coordinates the procedure of computing the statistics and returning the final result in plaintext to the user submitting the request. The processor interacts with all the relays producing statistics, the key authorities during the decryption step and is the only component which is accessible by the users. Crux needs at least one query processor running at all times, but more than one

processors may be in operation simultaneously. A fully functional query processor must be able to:

- Accept queries from users which request specific statistics computed on the Tor traffic and determine if the queried statistic is supported by Crux.
- Be able to sequentially request the decryption of a ciphertext from all key authorities, without revealing any information to the authorities.
- Forward the request to the Tor relays and specify the computation details (e.g. sketch size).
- Gather the statistics from the relays in an encrypted form and handle any transmission errors that may occur in the process.
- Aggregate the ciphertexts to form a single one which contains information about the overall traffic of Tor. In most cases, the sum of the ciphertexts is computed.
- Run the computation algorithm for the requested statistic and coordinate the decryption of the final and intermediate results.

The query processor is considered to be honest but curious. It can be merged with already operating Tor relays but in such a case the honest-but-curious assumption is extended to the relay as well.

4.1.4 User

The user is not a system component but more of an entity interacting with the system. The user is usually an network administrator who wants to extract some statistics for the network operation. It is able to submit queries to the query processor and then listens for the result of the computations. He has no access to any other system components. We assume that the user can be actively malicious, meaning that he can also submit specially crafted queries aiming to cause information leakage.

5 Prototype Implementation[1]

This section outlines the implementation of the prototype and provides details on the operation of the core components. We also discuss the tools and technologies we chose to use and provide information on their advantages and disadvantages.

5.1 Technologies

In our prototype all the components of the system are implemented in python. Python is robust and very flexible hence making it easy to incorporate with already existing systems and code. Additionally, it provides a wide-range of libraries which were useful for our purposes. One of the libraries used is *petlib* [32], which is an OpenSSL math and crypto wrapper, designed specifically for use in privacy enhancing technologies (PETs) projects. This means that it provides access to low level cryptographic functions but at the same time it abstracts away from purely technical details (e.g. memory management).

An implementation in C++ along with the inclusion of OpenSSL would perform slightly faster and would leave more room for performance optimizations. However, this would very likely result in relatively small improvements in performance, while the code complexity would be much higher. In the current work, code clarity is more important than optimum performance, and hence we choose to use python.

Moreover, for the communication between the components we choose to use JavaScript Object Notation (json). Json provides a straightforward and easy to parse way to encode data which is very useful when transferring data via network connections. However, Json is not the most efficient way to encode data in terms of space used, but in our use cases only a low volume of data is transferred. Hence we preferred the simplicity of json, even if it results in slightly increased data transfers.

5.2 Cryptosystem

In our deployment we use a cryptosystem based on a variant of El Gamal. More specifically, we implemented an Additively Homomorphic Elliptic-Curve cryptosystem which is

[1]The source code of the implementation can be found at: https://github.com/mavroudisv/Crux

a variant of El Gamal [10]. The cryptosystem comprises of the following three algorithms:

KeyGen(1^n): Given a security parameter n, elliptic curve E and a generator g are selected forming a group with order q. For the generation of the key pair, we choose $priv \in \mathbb{Z}_q$ and then we compute the public key as $pub = priv \cdot g$. The output consists of (E , g , pub) which is public and the private key $priv$.

Encryption(pub, m): Given the public key and a plaintext message m the algorithm computes the ciphertext $Ct = (A, B)$, where $A = k \cdot g$, $B = k \cdot pub + m \cdot g$ and $k \in \mathbb{Z}_q$ is randomly chosen.

Decryption(Ct, priv): During the decryption the two parts of the ciphertext and the private key are combined as such: $B - x \cdot A = m \cdot g$. This gives us $m \cdot g$ but not m itself. To extract m we need to solve a discrete logarithm problem, however, this is not computationally hard since the range of m is very limited and hence we can easily compute $i \cdot g$ for all the possible i's. Additionally in our implementation we optimized this procedure to reduce the runtime, by constructing a table with the discrete logarithms for all possible values. The keys in this table are the possible values of $i \cdot g$ and the values the corresponding i, this reduces the computation time to a dictionary lookup time, which is constant.

5.2.1 Common Public Key

One of main operations that each relay should be able to do is the computation of the *common encryption key*. The common encryption key is computed using the public keys released by the key authorities. More specifically, the relay first gathers those keys and subsequently combines them to produce the common key. Then the common key is used to encrypt all the data sent to the query processor. To do this we utilise the homomorphic properties of the keys:

Given a set of keys $K = \{pub_1, pub_2, ..., pub_n\}$, where n is the number of keys

we can compute the common public key $ckey = pub_1 + pub_2 + ... + pub_n = priv_1 \cdot g + priv_2 \cdot g + ... + priv_n \cdot g = (priv_1 + priv_2 + ... + priv_n) \cdot g$

From a theoretical perspective, the reason this works is that the public keys are points on the elliptic curve E and by definition the addition of two points of the curve, also returns a point on the curve.

Note that no authority holds the private key for the common public key, however there is a way to decrypt messages encrypted using this key, as seen in the following subsection.

5.2.2 Sequential Decryption

The sequential decryption takes place in the query processor and is used to decrypt the final or intermediate results while running the algorithms seen in section 5.4. As said, no entity holds the private key corresponding to the public common key. The decryption is done as follows:

Given a ciphertext $Ct = (A, B)$, encrypted with the common public key for the decryption we need to sequentially query all the key authorities requesting a partial decryption of the ciphertext. For instance, given the common public key $ckey = (priv_1 + priv_2) \cdot g$, constructed from only two authority keys and if $(A, B) = (k \cdot g,\ k \cdot (pub_1 + pub_2) + m \cdot g)$ is the ciphertext then the results of the decryption of the first authority would be $(A', B') = (k \cdot g,\ k \cdot pub_2 + m \cdot g)$ which can be fully decrypted by the second key authority. The above procedure is also shown in figure 5 with the difference that it features 3 authorities instead of 2.

The above procedure is using the decryption algorithm as a building block to realise the *threshold* feature of the cryptosystem we implemented. However, sequential decryption as presented here suffers from a security vulnerability, which was uncovered during the evaluation phase.s It's details and a fix are discussed in section 6.6.

5.2.3 Sum of Ciphertexts

In many of the algorithms which compute statistical values a sum of the measurements has to be computed. This is true for all three algorithms deployed in Crux (see also

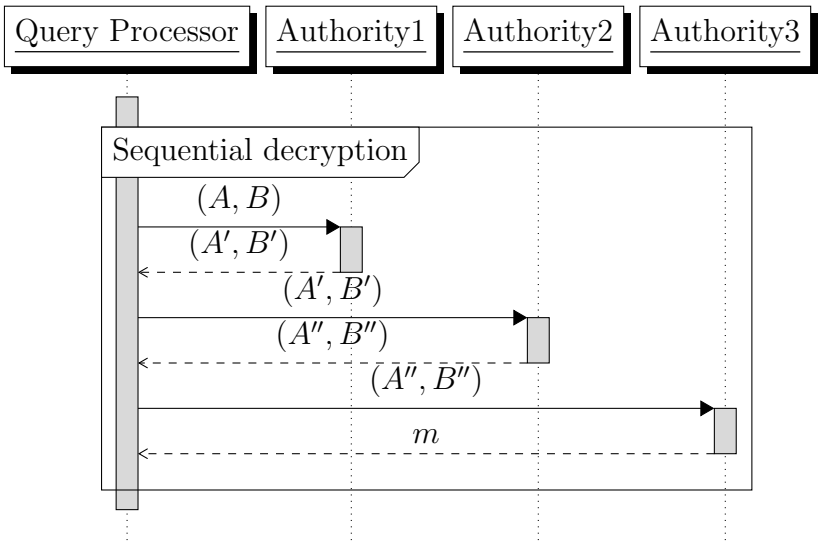

Figure 5: Sequential diagram of the sequential decryption protocol in a setup featuring only 3 key authorities.

section 5.4). However, the requirement in our case is that the sum has to be computed on encrypted values instead of plaintext values. In the case of Crux the chosen cryptosystem is additively homomorphic and hence we can easily compute the sum of two or more encrypted values (i.e. ciphertexts). More specifically, given two ciphertexts Ct_1 and Ct_2, which are encryptions of m_1 and m_2 respectively the addition operation is defined as follows:

$$Ct_3 = Ct_1 + Ct_2 \;\Rightarrow\; Enc_{pub}(m_3) \;=\; Enc_{pub}(m_1) + Enc_{pub}(m_2) \tag{4}$$

, where $m_3 = m_1 + m_2$

The proof for equation 4 is straightforward.

$$\begin{gathered} Ct_1 + Ct_2 = \\ Enc_{pub}(m_1) + Enc_{pub}(m_2) \;= \\ (A_1, B_1) + (A_2, B_2) \;= \\ (k_1 \cdot g, k_1 \cdot pub + m_1 \cdot g) + (k_2 \cdot g, k_2 \cdot pub + m_2 \cdot g) \;= \\ (k_1 \cdot g + k_2 \cdot g),\; (k_1 \cdot pub + m_1 \cdot g + k_2 \cdot pub + m_2 \cdot g) = \\ ((k_1 + k_2) \cdot g),\; ((k_1 + k_2) \cdot pub + (m_1 + m_2) \cdot g) \end{gathered} \tag{5}$$

which is a ciphertext with random value $k_1 + k_2$ and holds $m_1 + m_2$ which is equal to m_3. However, it should be noted that the addition operation is defined only if all the ciphertexts were generated using the same encryption key.

5.2.4 Sum of Sketches

In our design the Tor relays produce both simple and some more complex structures depending on the requested statistic. In particular, for the computation of the median (see also section 5.4.3) the relays return encrypted sketches of the traffic stream. The aggregation of these sketches is performed by the query processor. In our implementation we use linear sketches which can be sum-ed as long as they have the same dimension attributes (i.e. columns w and rows d) and use the same hash functions [25]. The sum of two sketches results in the sketch of the union of the two data streams these sketches we computed on.

In our case, the values of the sketches are encrypted. This requires a small modification of the aggregation procedure, in order to replace the plaintext addition with the addition operation of ciphertexts. Apart from this, the procedure is straightforward and the result is an encrypted sketch corresponding to the union of all the data streams observed by the relays.

5.3 Computation Procedure

Each time a request is sent by a user, a specific procedure takes place for the computation of the statistic requested.

5.3.1 Preliminary Computations

As seen in figure 6, initially each relay requests and receives the public key from all key authorities. This happens only once, unless a more complex mechanism for ephemeral keys is in place. For simplicity, here were assume that the authorities generate fresh keys only on startup and the keys do not get updated afterwards. When all the keys have been received by the relays and the common public key has been computed, the relays start to

listen for incoming requests.

5.3.2 Computation Core

The user sends a request to the query processor containing all the details needed for the computation. Each request should clearly define the type of statistic to be computed and the data on which it should be computed. The query processor receives the request and verifies that it is well-formed. Upon approval, it forwards it to the relays which it concerns. The relays receive the request, perform the necessary computations and encrypt the result. In our implementation, the result can be either a single value or a sketch. However, Crux can be easily extended to support other data types too. Then the encrypted result is send to the query processor.

The query processor then collects all the encrypted results from all the relays and aggregates them all (as seen in sections 5.2.3 and 5.2.4) to generate a result concerning the whole network. This is because, the statistics are to be computed on the traffic of the whole network (or a subset of the network) and not on individual nodes. The aggregation result is the input of the algorithm which computes the statistic requested by the user. Upon the completion of the algorithm execution the final result is returned to the user.

In the subsections following, we are going to examine the details of three such algorithms, which we implemented in Crux. Other algorithms can also be added if other statistics are needed.

5.4 Algorithms

We deployed three different algorithms which compute three distinct statistics. The execution of each of these algorithms is always coordinated by the query processor, while the input is provided by the Tor relays and any decryption needed is performed sequentially by the key authorities.

Below we present our algorithms in detail. However, please note that in the versions presented below the differentially private mechanisms have been removed, so that we can focus on the core of the algorithms. The noise added by those mechanisms in our

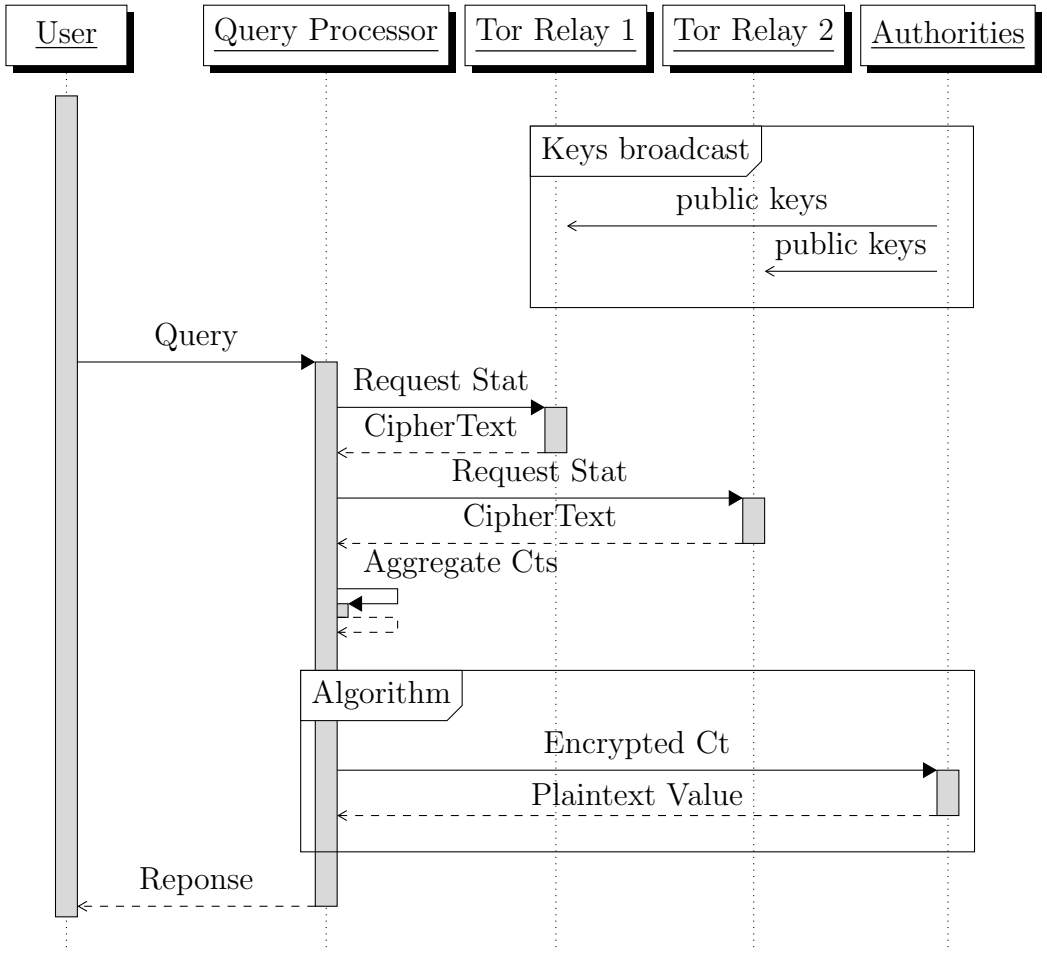

Figure 6: Sequential diagram providing an overview of the computation protocol.

implementation is added either in the intermediate results, or in the final one.

5.4.1 Mean

The input for the mean algorithm is a set of ciphertexts of the measurement of a specific Tor traffic characteristic as it was observed by each relay. The first step of the algorithm is to compute the sum of the given ciphertexts. The ciphertexts used for the computation of the mean hold a single value. The resulting sum is still encrypted and hence it is given to the decryption function, along with the list of authorities, and sequential decryption is performed. The result is divided with the number of relays and the mean is returned. The pseudocode of the mean algorithm can be seen in algorithm 1.

Algorithm 1: Algorithm for computing the mean value from a list of encrypted values without revealing individual values.

```
1 function Mean (lst_cts, lst_auths) Input  : Two lists containing the ciphertexts (lst_cts)
                                              and the IP addresses of the key authorities (lst_auths), respectively.
  Output: mean(lst_cts)
2 ct_sum ← Sum(lst_cts);
3 plain_sum ← decryption(ct_sum, lst_auths);
4 mean ← plain_sum)/|lst_cts|;
5 return mean;
```

5.4.2 Variance

This algorithm computes the variance of the values collected from the Tor relays. Its input is a list of encrypted values (lst_cts), a list of squares of the encrypted values (lst_cts_sq) and a list with the key authorities (lst_auths). Initially, the algorithm computes the sum of the measurements (ct_sum) and the sum of the squares of these measurements ct_sum_sqs and subsequently computes the mean of both (i.e. $mean$ and $mean\_sqs$). Then it squares the sum of the measurements ($mean\_sq$) and subtracts $mean\_sq$ from

mean_sqs to compute the variance. The pseudocode of the algorithm can be seen in algorithm 2.

Algorithm 2: The algorithm for computing the variance of the values provided by the Tor relays.

```
1  function Variance (lst_cts, lst_cts_sq, lst_auths);
   Input  : Three lists containing the measurements (lst_cts), their squares (lst_cts_sqs)
            and the IP addresses of the key authorities (lst_auths), respectively.
   Output: variance(lst_cts)
2  ct_sum_sqs ← Sum(lst_cts_sqs);
3  plain_sum_sqs ← decryption(ct_sum_sqs, lst_auths);
4  mean_sqs ← plain_sum_sqs/|lst_cts_sqs|;
5  ct_sum ← Sum(lst_cts);
6  plain_sum ← decryption(ct_sum, lst_auths);
7  mean ← plain_sum/|lst_cts|;
8  mean_sq ← mean * mean;
9  variance ← mean_sqs - mean_sq;
10 return variance;
```

5.4.3 Median

The algorithm computes the median of the values returned by the Tor relays, without observing any the individual values. To do this count-sketches are used as described in [21] and [22]. More specifically, the query processor gathers the sketches from all the relays and aggregates them to generate one sketch characterizing the overall network traffic. Then a divide and conquer algorithm is applied on this sketch to estimate the median. On each iteration of the algorithm the range is halved and the sum of all the elements on each half is computed. Depending on which half the median falls in, the range is updated and again halved until the range contains only one element. An assumption of our algorithm is that the range of the possible values is known. The pseudocode of the median algorithm can

be seen in algorithm 3, the original algorithm was proposed by Melis et al. in [64] and also can be found in [49].

6 Evaluation

This chapter performs a detailed examination and evaluation of our proposed design and its implementation. We focus on various areas of the system and perform a number of tests and experiments. In the first section, we discuss the methods we used to verify that the final implementation is complying with the system design and the security requirements. Then, we closely inspect our design in a number of areas and based on our findings we propose potential improvements on the scheme or the implementation.

6.1 Testbed

In order to properly evaluate Crux, we set up a testbed containing a number of components of each type. For the deployment of the testbed we chose to use Amazon Elastic Compute Cloud (Amazon EC2) which provides resizable computation capacity in the cloud. The benefit of EC2 is that it is fully customizable and hence we built systems which are similar with the ones expected to be used in a real-life deployment. More specifically, the specifications of the machines used for all components were: CPU Intel Xeon 3.3 Ghz, 1GB of memory, 10-15Mbps network bandwidth and a 10Gb magnetic hard disk. We set up 1 query processor, 3 key authorities and 4 clients.

Finally, to automate the administration and the experiments we used two python automation tools which are designed especially for EC2. The first one is boto [46] which provides an integrated interface to the services offered by Amazon Web Services and allowed us to automate the handling of our instances (e.g. start, update software). Fabric [44] was the second tool which we used. Fabric is a tool for streamlining the use of SSH for application deployment and administration tasks, which was very useful in our case as it enabled us to automate the execution of the Crux software components.

Algorithm 3: Algorithm for computing the median of the values provided by the Tor relays. The above algorithm is a slightly modified version of the algorithm introduced by Danezis et. al in [49] [64].

```
function Median (lst_cs, min_b, max_b, steps, auths);
  Input  : A set of encrypted sketches (lst_cs), the lower and higher bounds (min_b,
           max_b), the maximum number of possible iterations (steps) and the list of key
           authorities (auths).
  Output: median(lst_cs)
cs ← Sum(lst_cs);
L ← 0, R ← 0, bounds ← [min_b, max_b], total ← 'Undefined';
for i=0 to steps do
    old_bounds ← bounds;
    cand_median ← ⌊(bounds[0] + bounds[1])/2)⌋;
    if bounds[0] == cand_median then
        return decryption(cand_median, auths);
    end
    newl ← Sum([for i=bounds[0] to cand_median do Estimate(cs, i)]);
    if total == 'undefined' then
        newr ← Sum([for i=cand_median to bounds[1] do Estimate(cs, i)]);
        total ← newl + newr;
    else
        newr ← total - newl;
    end
    if newl + L > newr + R then
        R ← R + newr;
        bounds[1] ← cand_median;
        total ← newl;
    else
        L ← L + newl;
        bounds[0] ← cand_median;
        total ← newr;
    end
    if bounds == old_bounds then
        return decryption(cand_median, auths);
    end
end
```

Using this infrastructure and the aforementioned tools we conducted the experiments outlined in the following sections.

6.2 Quality Assurance & Verification

In order to confirm that Crux is indeed yielding the correct results and that every component is working as expected we developed a number of verification modules. In particular, we wrote unit tests which make sure that important parts of the system work according to the specifications seen on the previous chapters and a number of more complex tests which verify the proper interaction between more than one components. Some of these modules are automatically executed by the components (usually on startup) to make sure that all the sub-systems behave as expected, these are the *unit tests*. The unit tests are:

- *CipherText Unit Test*: This unit test makes sure that the implementation of the cryptosystem and consequently the produced ciphertexts support all the operations necessary for the computation of the statistics. More specifically, the test generates three public keys based on the predefined elliptic curve, encrypts a number of parameters, and then verifies that the operations of ciphertext addition, partial decryption (i.e. returns $m \cdot g$ without computing the logarithm) and full decryption work according to the specifications. This test is run once during the start up of each component. On failure, an error message is produced and the execution is halted.
- *Count-Sketch Unit Test*: This unit test verifies the implementation of sketches and their encryptions. During the test a new Count-Sketch is generated, encrypted and few elements are inserted. Then the *estimate* operation is applied on the encrypted sketch and the correctness of the results is verified. The test is ran only once during the start up of each component. On failure an error message is produced and the execution is halted.
- *Responsiveness Test*: This test is run by each component to make sure that the components and the interfaces it relies on are working properly. It is executed during the start up of each component and aims to verify the correct operation of the components by generating a random number, sending it to the tested component and the listening for the response. The test is successful if the response is received

within a pre-defined time-window and contains the random number increased by one. This procedure is ran during the start up of each component, and the default configuration of Crux allows five retries before returning an error. On failure an error message is produced and the execution is halted.

We also developed more complex tests which verify that the inter-component computations are yielding the correct results too. These tests are:

- *Decryption Correctness Test*: This test is complementary to the "CipherText Unit Test", seen earlier. This test is executed by the query processor and optionally by the user. Initially, a random value is encrypted using the public key of a key authority, and then the ciphertext is sent to this authority for decryption. The decryption result is compared with the initial value and if they do not match an error is thrown, otherwise the execution continues normally. Even though this test is useful during the development phase it is not necessary in a fully operational deployment, as other tests cover the same areas. Hence, in the default configuration it is disabled.
- *Full Operation Test*: This test verifies that all the components cooperate flawlessly and that the results yielded are correct. It involves all the components and is initiated by the user script on demand. At first, the user script sends a test request to the query processor. The request specifies that the differential privacy mechanisms should not be applied on the final result, which public dataset should be used for the computation of the statistics and the statistic to be computed. Subsequently, the query processor receives the request and handles it according to the protocols seen in sections 4 and 5. The relays are notified of the request and the public dataset which they should use (instead of the Tor traffic statistics they collect). Then the ciphertexts are returned to the query processor where the statistics are computed as seen in section 5.4, without the addition of noise (differential privacy mechanisms are disabled). Finally, the user receives the result and since it also has access to the public dataset used for the computations it recomputes the statistic and compares the two results.

In our implementation we used three public datasets from the London Datastore [6] [7] [71], however, any kind of dataset can be used as long as all the relays and users have access to it. Differential privacy is disabled, since it introduces random noise in the final result and hence the comparison of the results would always fail.

6.3 Features

Apart from the necessary mechanisms, Crux also includes a number of features which provide useful functionality and extend or improve aspects of the system. However, there are also some extra features which are not yet implemented. Table 1 provides a comprehensive list of these features and their implementation state, while the following list provides a more detailed explanation on each of them:

- *Parallelized Measurement Collection*: This feature enables the query processor to request and collect the measurements returned from the relays as fast as possible. More specifically, the query processor is able to send the requests and receive the responses in parallel, without waiting for the response before sending a new request. This approach eliminates the bottleneck which is due to the delay of each client while it computes the requested data. This feature is very useful for real-life applications where the number of relays is large and hence the total processing time adds up quickly. More formally, given n relays and assuming that the processing time for each is pt_i where $1 < i \leq n$, the naive approach would result in a total collection time equal with $\sum_{i=1}^{n}(pt_i)$, while the parallelized approach would take slightly more than $Max\{pt_1,\ pt_2,\ ...\ , pt_n\}$
- *Recovery from transmission errors*: The communication between the components is conducted through the internet and sometimes the network connections used suffer a quite high packet loss. For this reason, we deployed an additional communication layer on top of python's socket functions, which makes sure that all the packets are properly received. This proved very useful for the communication between the relays and the query processor, as a significant volume of encrypted and serialized objects is transmitted and even small delays in the arrival of the segmented packets resulted in incomplete transmissions and failures in the reconstruction of the object.

- *Analysis & Debugging*: In order to be able to fully analyse the behaviour of each component we also incorporated some profilers in the source code of all components except from the user's script. We choose two well-documented and commonly used profilers: cProfile and LineProfiler. cProfile analyses how python consumes processor resources and provides information such as the runtime of each function or script and the number of calls it makes. LineProfiler provides a more detailed view as it reports the resource usage statistics of each source code line. Our implementation also supports a call graph library (i.e. PyCallGraph) which visualizes the function calls and the parent-child relationships between the different functions.
- *Avoid Leakage in sequential decryption*: As described in section 5.2.2 during the sequential decryption is last authority to decrypt returns the plaintext to the query processor. However, this means that the authority which is assumed to be honest-but-curious can gather information since it sees both the final result and the intermediate ones. To avoid this, we added an additional layer of encryption, which makes sure that the authorities cannot observe anything meaningful. This vulnerability was found in the evaluation phase and it is discussed in more detail in subsection 6.6.
- *Randomization of query processor key*: As an extension to the above feature, the keys used for the encryption are freshly derived each time a new decryption sequence is initiated.
- *Ephemeral keys and Signed Keys*: These features are not yet implemented and thus they are discussed in detail in chapter 7.

6.4 Computation Accuracy

It is important to investigate how the different operation parameters affect the accuracy and the performance of the system. One of the most important parameters, is the size of the sketches. The size of the sketches is determined by two parameters, the number of hash functions used (i.e. the d parameter) and the number of bins which each row of the sketch has (i.e. the w parameter). In our implementation, the sketch is used to estimate only one statistical value, the median. For this reason, our experiment is based on this statistic.

Feature	Implemented	Component
Parallelized measurement collection	✓	Q. Processor
Recover from transmission errors	✓	All
Analysis & Debugging	✓	All except user
Avoid leakage during decryption	✓	Q. Processor
Randomization of q. processor key	✓	Q. Processor
Ephemeral keys	✗	Authorities
Signed keys	✗	Authorities

Table 1: Table listing the features which are available in the current version of Crux and some additional features which where not implemented yet.

The experiment was conducted as follows: Our setup included one query processor, one authority and one relay. In normal conditions more than one authorities and relays would be available but for the specific experiment the number of participants does not affect the final result. We then selected a set of numbers from the London Datastore [6] [7] [71] and provided this set to the relay as input. We selected these datasets as they contain real-life data and they are very similar with those that Tor relays collect. We then sent multiple requests to the query processor for the computation of the "median" for this set. Each request included different sketch parameters ranging from 1 to 10 for the number of hashes (i.e. the number of sketch rows) and from 3 to 70 for the number of bins in each row of the sketch (i.e. the sketch columns). After each request was served we stored:

1. The sketch parameters (i.e. dimensions)
2. The discrepancy from the correct median
3. The total runtime

We used these data to derive two datasets one containing the errors in the result, and one containing the runtime both in relation to the sketch parameters. From a preliminary analysis of the data we found that the number of columns of the sketch does not significantly affect the estimation accuracy, as long as the number of columns remain within a meaningful range. This conclusion was considered during the visualization of the first dataset, which is shown in figure 7. In the figure we observe that the count-sketches with

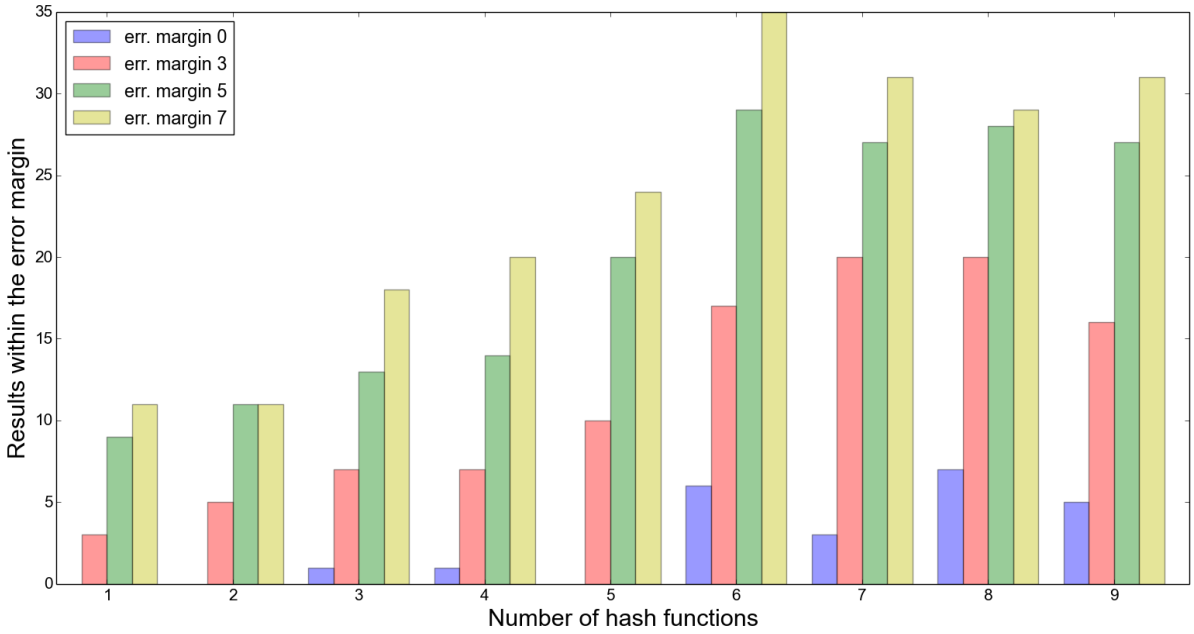

Figure 7: Bar-chart demonstrating the correlation between the dimensions of the count-sketch and the error in the estimation of the median. The height of each bar indicates the number of results within the given error margin for the specific number of rows (i.e. hash functions).

a small number of rows (i.e. < 3) are not able to provide accurate estimations regardless of the number of columns. Additionally, we see a steady increase in the accuracy for all error margins, as the number of rows increases. This increase appears to stabilize when six rows are used, as any further increase does not improve the results. Instead, the sketch with six rows seems to perform sightly better than the sketches with more rows, however, since the differences are not very large for the results within smaller error margins we are inclined to partially attribute this to the specific data. All in all, it seems that a sketch with six rows outperforms sketches with different number of rows.

The second dataset is visualized in figure 8. From the graph we conclude that the number of columns does not significantly affect the runtime of the estimation algorithm and this is in accordance with our preliminary finding from the errors dataset. However, the number of rows influence the total runtime. The relationship appears to be linear, which can be explained if we consider that the lookup time in a (hashed) dictionary is constant and in

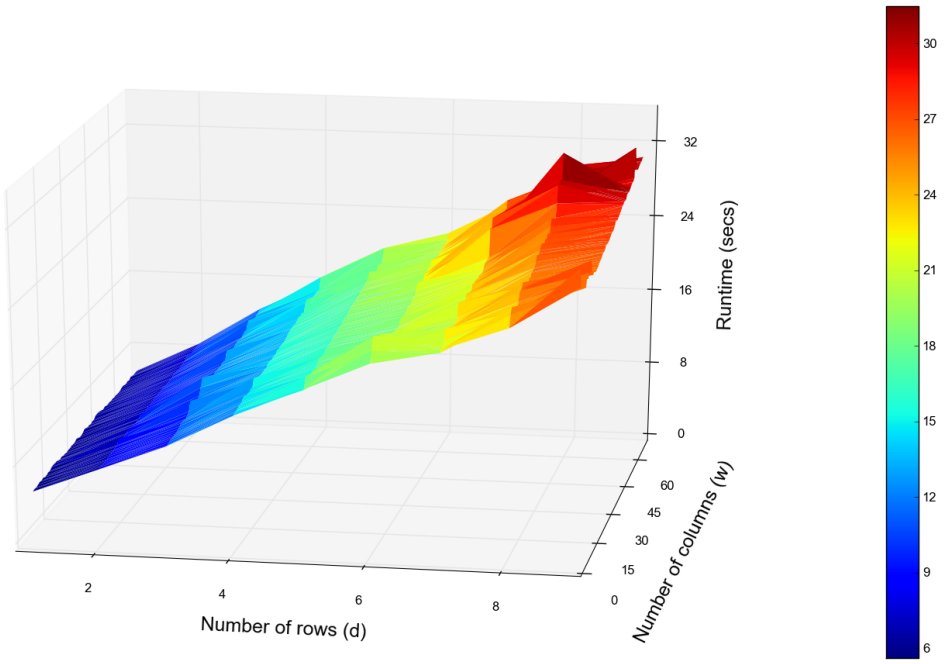

Figure 8: 3D surface plot which demonstrates how the computation time increases as the dimensions of the count-sketch are growing.

our case the number of hash lookups grows linearly. Hence, any increase in the number of rows comes with a cost on the runtime that the administrator should consider when determining the value of the parameter.

Additionally, from the previous two figures we also conclude that there is no practical gain in using very high number of rows in the sketches, as such a configuration would increase the runtime without necessarily improving the accuracy. We should also note that throughout this experiment the differential privacy mechanism was disabled so that it wont affect the final results.

We also conducted experiments to evaluate the accuracy of the mean and the variance statistics however, the results indicated that both methods achieve nearly 100% accuracy. This is understandable, since they rely only on ciphertexts and at no point during the computation, lossy methods are used (e.g. hashing). Any discrepancies observed are always negligible and can be attributed to the loss of accuracy in floating point operations.

6.5 Communication Overhead

An important aspect of the implemented system is the network usage and the bandwidth that the component interaction requires. In the case of a system which transfers large amounts of data even small inefficiencies in the implementation can result in exponential increases in the network usage. This is also true for our system, since it is expected to work with a large number of relays. In this section we run experiments to make sure that Crux is as efficient as possible, that it does not create redundant connections and to investigate the distribution of the bandwidth usage between the components.

For the experiments we used DarkStat [65] and TCPdump [54], which are both network usage monitoring tools based on the network capture library *libpcap* [55]. DarkStat offers various features, and one of them is to capture the traffic of each port and then provide a report of the volume transferred. TCPdump is able to capture the whole tcp conversations and remove any noise from other applications with the use of its port filters. Using these tools, we measured the bandwidth usage in 4 different computation phases: on startup (table 2), during the computation of the median (table 3), when computing the mean (table 4), and when computing the variance (table 5). Our testbed setup included one relay, one query processor and one key authority. We decided to use one component of each type so that we can measure the network load of each type of component more accurately.

Our first experiment measured the load during the initialization phase of the components (table 2). As expected the initialization procedure is very light and it doesn't add a significant burden to any of the components. The number of SYNs received verify that during this procedure the key authority receives two ping requests from the relay and the query processor and one request for its public key by the relay.

We also conducted three more experiments, examining the network usage during the computation of the mean, median and variance. The results of these experiments can be seen in the tables 4, 3 and 5 respectively. Our measurements gave identical results.

Component	Tx (Bytes)	Rx (Bytes)	Total (Bytes)	SYNs Received
Key Authority	1565	1795	3360	3
Query Processor	320	288	1500	1
Relay	1145	1132	2277	0
User	0	0	0	0

Table 2: Network activity during initialization

Component	Tx (Bytes)	Rx (Bytes)	Total (Bytes)	SYNs Received
Key Authority	8.280	13.758	22.038	20
Query Processor	19.800	150.709	170.509	1
Relay	138.153	5.541	143.694	1
User	1062	447	1509	0

Table 3: Network activity during the computation of median

Component	Tx (Bytes)	Rx (Bytes)	Total (Bytes)	SYNs Received
Key Authority	414	689	1103	1
Query Processor	16.511	1.419.613	1.436.124	1
Relay	1.430.877	15.312	1.446.189	1
User	715	510	1225	0

Table 4: Network activity during the computation of mean

Component	Tx (Bytes)	Rx (Bytes)	Total (Bytes)	SYNs Received
Key Authority	3.312	6192	5.512	2
Query Processor	69.199	2.906.383	2.975.582	1
Relay	2.810.744	63.107	2.873.851	2
User	944	460	1404	0

Table 5: Network activity during the computation of variance

More specifically, in all the experiments the query processor is the component which has the highest volume of input data. This is expected since it collects the data from all the relays. It is also interesting that in the mean and variance experiments the rx volume of

the query processor is much higher compared to the median. This is because in those two statistics we compute the sum of our measurements and this results in very large numbers, whereas the sketch stores much smaller numbers.

On the other hand, the relay has a very high output (tx) volume, which is roughly equal to the input of the query processor. However, we expect that this high output volume will decrease as the number of relays grows. More specifically, in our experiment the relay returned the whole dataset instead of a single measurement. In a real-life scenario each relay will return only one encrypted measurement and this would greatly reduce the output volume. Another interesting fact is that the number of SYN packets received in these tables depends fully on the computation algorithm used. In particular, in the case of the mean only one sequential decryption is initiated and hence only one SYN packet was received. However, the variance algorithm requires two decryptions and the median one requires one decryption for each iteration, as seen in algorithm 3.

It is also very positive that the user script has very low input and output volumes meaning the the end-user can still use the system even if he has a connection with very limited bandwidth. Finally, we should also note that in an ideal scenario the Tx and Rx columns on each of the above tables would add up to the exact same number. However, due to lost packets, retransmissions etc the sum of the numbers in those columns are similar but not exactly the same.

6.6 Security Considerations & Potential Attacks

In this part of the evaluation we carefully examined all the algorithms and the protection mechanisms used by Crux and verified that they provide adequate protection from all the adversaries included in the threat model. During this procedure we found a few potential attacks and based on our findings devised fixes where it was necessary.

More specifically, we found two attacks. The first one can be executed by an attacker who falls within the threat model. More specifically, the assumption is that the key au-

thorities are honest but curious, meaning that do not actively attack the system but they try to extract as much information by observing the traffic that goes through them. Additionally, we assumed that the attacker is able to observe the packets that are transmitted between the components to collect information.

Sequential Decryption Leak: In our analysis we observed that one of the procedures which may leak information is sequential decryption. In the initial iplementation of the sequential decryption, the intermediate results were transferred in plaintext and hence an eavesdropping adversary could gradually learn information about the dataset by submitting specially crafted requests and then observing the final and intermediate results. A naive solution to this problem, would be to have the key authority encrypt the intermediate results with the public key of the query processor. Even though, this would indeed provide protection from an eavesdropping adversary, it would still allow the last authority of the sequential decryption to observe the intermediate results. A much better solution would be the query processor to encrypt the ciphertext with a key (based on the same cryptosystem as the keys of the authorities) and then decrypt the result returned by the decryption authority. This solution exploits the homomorphic properties of the cryptosystem and enables the query processor to act as a temporary authority to protect the confidentiality of the intermediate results. In our current implementation, this mechanism has already been deployed.

The examination also uncovered one attack which can seriously disrupt the system. Even though, the adversary needs to have some additional abilities which are not included in the Crux's or Tor's threat model, we decided to include it because of its severity. This adversary aims to uncover the traffic measurements observed by a single Tor relay. For this attack, the adversary is also assumed to be able to tamper with the packets transferred between the components and hence he is able to perform a man-in-the-middle attack and replace the transmitted data with maliciously crafted data. However, the adversary is not able to interfere with the computations on any of the component neither has access to non-public data.

Integrity attack on sketches: Initially, the attacker collects the public keys of the key au-

thorities and computes the common key. He then encrypts using this key $n-1$ (where n is the number of relays participating in the computation) values or count-sketches and stores a ciphertext of each of them. He then waits for a stat request to arrive or submits one himself, and once the relays attempt to return their encrypted measurements to the query processor, he intercepts the packets of *all the but one* relays and replaces them with the ciphertexts computed earlier. Then query processor collects the measurements and executes the computation protocol normally, as it cannot distinguish the malformed messages. Then the adversary collects the final result and it is trivial to infer the value contributed by the relay of interest, since he is aware of all the other values of the computation and the final result. In the case of the median, since the count-sketches a lossy, the adversary will be only able to only closely approximate the sketch contributed by the relay.

The fix for this attack is straightforward, all the components must hold a certificate issued by a trusted authority and use it to sign all the messages they sent. This would protect the integrity of the communication and hence this attack would be no longer possible.

7 Conclusions and Future directions

This thesis introduced Crux a system enabling the computation of privacy-preserving statistics for the Tor network. In the last year, the Tor development team has taken substantial steps to enable the collection of traffic measurement by the network relays. However, the problem of aggregating the values from all the network relays and computing of the final result without jeopardizing the user's anonymity, has not been fully tackled yet.

Crux provides a solution to this problem. To achieve this, it utilizes an additively homomorphic encryption scheme to aggregate the measurements of the Tor relays and then executes a computation protocol to derive the value of the statistic. In this work, we initially discussed the privacy problems currently faced in data sharing and processing, and examined existing works on privacy-preserving computations and statistic computations. We also reviewed the operation of the Tor network and the concept of sketches. Based on

our findings, the threat model and the requirements for Crux were presented, and then the system design was built to satisfy them. Subsequently, we outlined the implementation details of Crux. The implementation follows the system design and provides a modular solution which could be applied on Tor with minimal effort. Its performance both in terms of runtime and in terms of security is examined in the last chapter on which we used a variety of methods and tools to evaluate Crux. The evaluation covers all aspects of the system and resulted in a variety of findings which would enable the administrators to achieve the optimal performance depending on their requirements. Additionally, during the evaluation we identified a number of shortcomings of our approach and proposed potential improvements and new features.

Future work in the area could incorporate these features and combine them with new promising research directions. For instance, the recent work on persistent data sketching [76] shows potential for the development of algorithms and systems which provide statistics within a user-defined time window. Another direction showing potential, is the parallelization of the sequential decryption protocol which would greatly reduce the runtime of the whole procedure and poses an interesting mathematical challenge. Moreover, there are some features (as seen in section 6.3) which are not yet implemented but they are very interesting as they would further improve the capabilities of Crux. One such feature is the *ephemeral keys*. This feature enables the key authorities to renew their encryption key pairs based on either the time elapsed since its release (e.g. once per day) or after a predefined number of decryption requests. Both methods, aim to hinder cryptanalytic attacks by preventing the collection of a vast number of ciphertexts produced with the same key. However, one of the reasons that this feature is not implemented yet is that additional mechanisms are required in order to handle cases where the keys are renewed while processing a decryption request. Additionally, the feature may be exploited by an attacker to cause a Denial of Service by using the available decryption quota before a benign user's decryption is completed. Another such feature is *signed keys*. In our threat model we assumed that an attacker can observe the communication channels but he cannot tamper with the traffic. This allowed the key authorities to simply publish their keys.

However, if an adversary is able to conduct a man in the middle attack he could then trick the relays to use his keys instead of those of the authorities. To counter such an attack the public keys could be signed using a certificate by a trusted authority (e.g. verisign). This feature is not implemented yet but it would increase the level of security and would allow us to relax the security assumptions even more.

In conclusion, Crux is one of the first systems which use privacy preserving schemes to compute statistics for Tor. It is modular and provides all the necessary mechanisms to remain safe, efficient and adaptable to future needs.

References

[1] Torstatus. `http://torstatus.blutmagie.de/index.php`. Accessed: 2015-08-30.

[2] Ben Adida. Helios: Web-based open-audit voting.

[3] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer Science & Business Media, 2007.

[4] Noga Alon, Phillip B Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 10–20. ACM, 1999.

[5] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[6] Lower Super Output Area in Greater London Authority. London datastore. 2011. *Link: http://data.london.gov.uk/dataset/msoa-atlas.*

[7] Middle Super Output Area in Greater London Authority. London datastore. 2011. *Link: http://data.london.gov.uk/dataset/lsoa-atlas.*

[8] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 633–634. Society for Industrial and Applied Mathematics, 2002.

[9] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM, 2008.

[10] Josh Benaloh. Dense probabilistic encryption.

[11] David Bernhard, Véronique Cortier, Olivier Pereira, Ben Smyth, and Bogdan Warinschi. Adapting helios for provable ballot privacy. In *Computer Security–ESORICS 2011*, pages 335–354. Springer, 2011.

[12] George Robert Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*, pages 313–313. IEEE Computer Society, 1899.

[13] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security-ESORICS 2008*, pages 192–206. Springer, 2008.

[14] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[15] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography and Data Security*, pages 142–147. Springer, 2006.

[16] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of cryptography*, pages 325–341. Springer, 2005.

[17] Michael Brenner, Jan Wiebelitz, Gabriele von Voigt, and Matthew Smith. Secret program execution in the cloud applying homomorphic encryption. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, pages 114–119. IEEE, 2011.

[18] Philipp Burtyka and Oleg Makarevich. Symmetric fully homomorphic encryption using decidable matrix equations. In *Proceedings of the 7th International Conference on Security of Information and Networks*, page 186. ACM, 2014.

[19] Claude Castelluccia, Einar Mykletun, and Gene Tsudik. Efficient aggregation of encrypted data in wireless sensor networks. In *Mobile and Ubiquitous Systems: Networking and Services, 2005. MobiQuitous 2005. The Second Annual International Conference on*, pages 109–117. IEEE, 2005.

[20] T-H Hubert Chan, Elaine Shi, and Dawn Song. Privacy-preserving stream aggregation with fault tolerance. In *Financial Cryptography and Data Security*, pages 200–214. Springer, 2012.

[21] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*, pages 693–703. Springer, 2002.

[22] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[23] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[24] European Commission. Regulation of the european parliament and of the council. *On the protection of individuals with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*, jan 2012.

[25] Graham Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.

[26] Graham Cormode and Minos Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*, pages 13–24. VLDB Endowment, 2005.

[27] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[28] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[29] Graham Cormode, Cecilia Procopiuc, Divesh Srivastava, and Thanh TL Tran. Differentially private summaries for sparse data. In *Proceedings of the 15th International Conference on Database Theory*, pages 299–311. ACM, 2012.

[30] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits. In *Computer Security–ESORICS 2013*, pages 1–18. Springer, 2013.

[31] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*, pages 643–662. Springer, 2012.

[32] G. Danezis. Privacy enhancing technolgies library. `https://github.com/gdanezis/petlib`, 2015.

[33] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type iii anonymous remailer protocol. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 2–15. IEEE, 2003.

[34] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security - SEGS '13*, pages 75–80, New York, New York, USA, November 2013. ACM Press.

[35] Dorothy E Denning, Peter J Denning, and Mayer D Schwartz. The tracker: A threat to statistical database security. *ACM Transactions on Database Systems (TODS)*, 4(1):76–96, 1979.

[36] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[37] Cynthia Dwork. Differential privacy: A survey of results. In *Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[38] Cynthia Dwork. Differential privacy. In *Encyclopedia of Cryptography and Security*, pages 338–340. Springer, 2011.

[39] Zekeriya Erkin and Gene Tsudik. Private computation of spatial and temporal power consumption with smart meters. In *Applied Cryptography and Network Security*, pages 561–577. Springer, 2012.

[40] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1054–1067. ACM, 2014.

[41] Philippe Flajolet and G Nigel Martin. Probabilistic counting. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 76–82. IEEE, 1983.

[42] Philippe Flajolet and G Nigel Martin. Probabilistic counting algorithms for database applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[43] Caroline Fontaine and Fabien Galand. A Survey of Homomorphic Encryption for Nonspecialists. *EURASIP Journal on Information Security*, 2007:1–10, January 2007.

[44] Jeff Forcier. Fabric: A tool for streamlining the use of ssh for application deployment.

[45] Arik Friedman and Assaf Schuster. Data mining with differential privacy. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 493–502. ACM, 2010.

[46] Mitch Garnaat. Boto: Python interface to amazon web services.

[47] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[48] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[49] Danezis George. Tormedian, petlib, 2015.

[50] Phillip B Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[51] Phillip B Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, volume 97, pages 466–475, 1997.

[52] David Goulet, Aaron Johnson, George Kadianakis, and Karsten Loesing. Hidden-service statistics reported by relays.

[53] Mario Heiderich, Tilman Frosch, Marcus Niemietz, and Jörg Schwenk. The bug that made me president a browser-and web-security case study on helios voting. In *E-voting and identity*, pages 89–103. Springer, 2012.

[54] V Jacobsen, Craig Leres, and Steven McCanne. Tcpdump/libpcap, 2005.

[55] V Jacobson, C Leres, and S McCanne. libpcap, lawrence berkeley laboratory, berkeley, ca. *Initial public release June*, 1994.

[56] Marek Jawurek and Florian Kerschbaum. Fault-tolerant privacy-preserving statistics. In *Privacy Enhancing Technologies*, pages 221–238. Springer, 2012.

[57] Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 549–560. ACM, 2013.

[58] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *Privacy Enhancing Technologies*, pages 175–191. Springer, 2011.

[59] I. Leontiadis, R. Molva, and M. Onen. Privacy preserving statistics in the smart grid. In *Distributed Computing Systems Workshops (ICDCSW), 2014 IEEE 34th International Conference on*, pages 182–187, June 2014.

[60] Yi Li, Huy L Nguyen, and David P Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 174–183. ACM, 2014.

[61] Yossi Matias. Dynamic maintenance of wavelet-based histograms. Citeseer.

[62] Vasileios Mavroudis. Literature review on privacy-preserving computations. *Introduction to Research*, 2015.

[63] Frank McSherry and Ilya Mironov. Differentially private recommender systems: building privacy into the net. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 627–636. ACM, 2009.

[64] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches, 2015.

[65] Emil Mikulic. Darkstat: Captures network traffic, calculates statistics about usage, and serves reports over http.

[66] Ulf Möller, Lance Cottrell, Peter Palfrader, and Len Sassaman. Mixmaster protocolversion 2. *Draft, July*, 2003.

[67] Shanmugavelayutham Muthukrishnan. *Data streams: Algorithms and applications*. Now Publishers Inc, 2005.

[68] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[69] Yael Onn, Michael Geva, Y Druckman, A Zyssman, R Lev Timor, et al. Privacy in the digital environment. *Haifa Center of Law & Technology*, pages 1–12, 2005.

[70] Claudio Orlandi. Is multiparty computation any good in practice? In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5848–5851. IEEE, 2011.

[71] Ward Profiles and Atlas. London datastore. 2011. *Link: http://data.london.gov.uk/dataset/ward-profiles-and-atlas*.

[72] Monika R Henzinger Prabhakar Raghavan. Computing on data streams. In *External Memory Algorithms: DIMACS Workshop External Memory and Visualization, May 20-22, 1998*, volume 50, page 107. American Mathematical Soc., 1999.

[73] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[74] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[75] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[76] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 795–810, New York, NY, USA, 2015. ACM.

[77] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[78] Andrew C Yao. Protocols for secure computations. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE, 1982.