Enhancing the Onion Name System for Darknet

G. A. A. Maduranga

Enhancing the Onion Name System for Darknet

G. A. A. Maduranga
Index No: 13000683

Supervisor: Dr T. N. K. De Zoysa

December 2017

Submitted in partial fulfilment of the requirements of the
B.Sc. in Computer Science Final Year Project (SCS4124)

Declaration

I certify that this dissertation does not incorporate, without acknowledgement, any material previously submitted for a degree or diploma in any university and to the best of my knowledge and belief, it does not contain any material previously published or written by another person or myself except where due reference is made in the text. I also hereby give consent for my dissertation, if accepted, be made available for photocopying and for interlibrary loans, and for the title and abstract to be made available to outside organizations.

Candidate Name: G. A. A. Maduranga

……………………………………………
Signature of Candidate   Date:

This is to certify that this dissertation is based on the work of

Mr G. A. A. Maduranga

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Supervisor Name: Dr T. N. K. De Zoysa

……………………………………………
Signature of Supervisor   Date:

Supervisor Name: Mr T. G. A. S. M. De Silva

……………………………………………
Signature of Supervisor   Date:

Abstract

The surface net as a whole does not provide the capabilities for an individual to share all kinds of information without having to take responsibility for the content posted by the individual. In a place where surface net fails, darknet prospers. Individuals with the need to share information with plausible deniability use darknet to achieve this end of the goal. However, the usability issues of the darknet act as a continuous hindrance to the use and growth of darknet. Thus, there is a requirement for a secure, decentralized name system for darknet, to overcome the above issue.

In this document, the Onion Name System (OnioNS) is considered as a possible candidate to overcome the above-mentioned issues. Primarily the security concerns OnioNS introduces to the Tor network is analysed. Special concern is given to the possibility of time analyses attacks that could be carried out on the Tor network due to restrictions imposed by the OnioNS. The concerns raised are addressed in this dissertation and a method to overcome them is introduced. A novel hash tree data structure is introduced as the core component of the proposed solution. The proposed solution is analysed in order to guarantee that it is capable of implementing all the features of the OnioNS while minimizing the security threats the existing system imposes on Tor network.

Further analysis of the novel solution is discussed in order to identify if the solution introduced has implemented any additional vulnerabilities to the Onion Name System or the Tor network.

Preface

The research idea originated from my interest to learn about and contribute to the network and information security domain. The contents that are taken from works other than of my own are referenced and acknowledged accordingly. The code snippets provided in the section five as well as appendix A, are works of myself. The implementation of the solution, result analysis and was carried out under the supervision of my supervisor.

Acknowledgement

I would like to express my sincere gratitude to my research supervisor, Dr T. N. K. De Zoysa, senior lecturer at University of Colombo School of Computing and my co-supervisor, Mr T. G. A. S. M. De Silva, assistant lecturer at University of Colombo School of Computing, for providing immense support and guidance throughout the research.

I would like to extend my thanks to Mr P. Wijesekara, Research Scholar at the University of California, Berkley, and Mr. Jesse Victors, a contributor to the Tor Project, for providing me with content and suggestions, through the period to continuously improve the research carried out. Special thanks must be given out Mr P. N. Pieris, Mr T. Ranathunga, Mr V. D. Liyanage, Mr S. S. K. Malkakulage and Mr T. Deshan for helping out with various aspects of the research. I also would like to express my gratitude to Dr J. S. Goonatilake and Dr C. K. Keppitiyagama for the feedback and evaluation provided from the start of the research. Special thanks to Dr H. E. M. H. B. Ekanayake for the assistance provided as the computer science project coordinator of the final year.

Further, I would also like to acknowledge all the support provided by friends and family members to complete this research as it is. As a final note, I would like to thank and recognize the contribution of all the people that helped me in any regard, to complete this research.

Table of Contents

Declaration ..... **i**

Abstract ..... **ii**

Preface ..... **iii**

Acknowledgement ..... **iv**

Table of Contents ..... **v**

List of Figures ..... **vii**

List of Tables ..... **viii**

List of Acronyms ..... **ix**

Chapter 1 - Introduction **1**

1.1 Background to the Research 2

1.2 Research Problem and Research Questions 4

1.3 Justification for the research 4

1.4 Methodology 6

1.5 Outline of the Dissertation 7

1.6 Delimitations of Scope 7

1.7 Summary 8

Chapter 2 - Literature Review **9**

2.1 Introduction 9

2.2 Approaches for Darknet Naming Systems 9

Chapter 3 - Design **15**

3.1 Introduction 15

3.2 Research Design 15

3.3 Time Window Limitation Analysis 16

3.4 Hash-able data structures 18

3.5 Authenticated Skip List 18

3.6 B+ Hash Tree 19

3.7 Summary 23

Chapter 4 - Implementation 24

4.1 Introduction 24

4.2 Software Tools 24

4.3 Implementation Details 24

4.4 Summary 27

Chapter 5 - Results and Evaluation 28

5.1 Introduction 28

5.2 Results 28

5.3 Summary 32

Chapter 6 - Conclusions 33

6.1 Introduction 33

6.2 Conclusions about research questions 33

6.3 Conclusions about research problem 35

6.4 Limitations 35

6.5 Implications for further research 36

References 37

Appendix A: Code Listings 39

List of Figures

Figure 1.1: Zooko's triangle 3
Figure 1.2: Proposed research methodology 7
Figure 2.1: High-level data flow diagram in OnioNS [4] 13
Figure 3.1: High-level design diagram 16
Figure 3.2: Authenticated skip list [21] 19
Figure 3.3: B+ hash tree design 20
Figure 3.4: Insertion Algorithm 21
Figure 3.5: Split_Interior Function 22
Figure 3.6: Verification Algorithm 22
Figure 5.1: Initialization time analysis 30
Figure 5.2: Insertion time analysis 31
Figure 6.1: Build errors 36

List of Tables

Table 5.1: Tree initialization time comparison 29

Table 5.2: Node insertion time comparison 30

List of Acronyms

UTC	–	Universal Time Coordinated
ISP	–	Internet Service Provider
OnioNS	–	Onion Name System
OnionDNS	–	Onion Domain Name System
GNS	–	GNU Name System
GNU	–	GNU’s Not Unix

Chapter 1 - Introduction

With the growth of the Internet and other means of communication, the online privacy and security of an average person are continuously put to the test. Therefore, with the increasing risks, the development of privacy-enhancing technologies has also increased. Though there have been various attempts, on the surface web, the anonymity guarantees are not given to the user. As the underlying UDP, TCP or IP protocols cannot hide their headers, the routing information between two parties on the traditional Internet, is visible to other parties.

Following the growing concerns about privacy and anonymity, the users have been increasingly turning towards privacy enhancing technologies, in recent years. As an out of the box solutions, users tend to use proxies and VPNs primarily to enhance their privacy. As these tools, itself keeps track of users; they cannot be taken as better tools to enhance privacy [1].

Most of the anonymity tools that are in use today, are descendants of the early mixnets [2], that were invented in the early 1980s. In a mixnet, the user traffic is scrambled, delayed, retransmitted and partially decrypted, before being received at the final destination. This heavily obscures the correlation between the source and the destination of the traffic. Therefore, it adds anonymity and privacy, to a certain level, when exchanging network traffic. Tor is a descendant of early mixnets, which uses a three-hop circuit between a server and a client, to provide privacy and anonymity, to both the client and server. However, Tor suffers from a particular flaw since its inception. That is, any hidden service that is in Tor is given a hash value for a domain, instead of a memorable human name. Though this has not limited the popularity of Tor, it could be seen that, if a more DNS like human memorable naming system could make Tor much user-friendlier. There have been more than a few approaches to make this a reality, such as OnionDNS [3] and OnioNS [4], which are described in detail, in the literature review.

1.1 Background to the Research

The Onion Router – Tor [5] is currently the most popular low-latency onion routing system which provides privacy and anonymity for the users. The services provided by Tor are mainly of two folds. One being, providing mechanisms for a regular user to browse the surface net with anonymity guarantees. Second being, Tor providing access to a unique set of websites in the Darknet known as Hidden Services or Onion Services. Recent studies show that Tor provides the users access to over 45,000 hidden services on average [6].

Hidden services are provided in such a way to mask the service's IP address from the users. To access an onion service, the service should be referenced by the first 16 characters of the SHA-1 hash of the service's public RSA key. This reference is then appended with a '.onion' pseudo-top level domain to complete the onion address of the service. The constructed onion address is equivalent to a URL that is used in the surface net. Therefore, an address constructed through the above mechanism could be used to access the service. The usage of the service's public key makes it possible to confirm the one-to-one relationship between the service and the provided address of the service. '32rfckwuorlf4dlv.onion' is a hidden service address which points to Onion URL Repository. This is an example for the 16-character hash address generated for hidden service reference. It is abundantly clear that such an address is not memorable.

Even though such an address is not human-meaningful, by providing such an address, Tor achieves the ability to provide security and decentralization. In this context, security refers to having an abundantly large collision free address space, which in turn guarantees that each address is unique in the tor network. Decentralization allows the tor network to be resistant to various security risks and censorship attempts against the network.

In 2001, Zooko Wilcox-O'Hearn proposed a conjecture which came to know as Zooko's triangle [7]. Figure 1.1 shows the Zooko's triangle. It stated that a persistent

naming system could only achieve two at a time of three properties, namely, Human-meaningfulness, Decentralization, and Security. This model provides a general explanation of why Tor addresses are not capable of achieving human-meaningful names, since its inception in 2002. However, this is not the case since, in the recent years, several models that exhibit all three desirable features of a naming system has been modelled [8] [4].

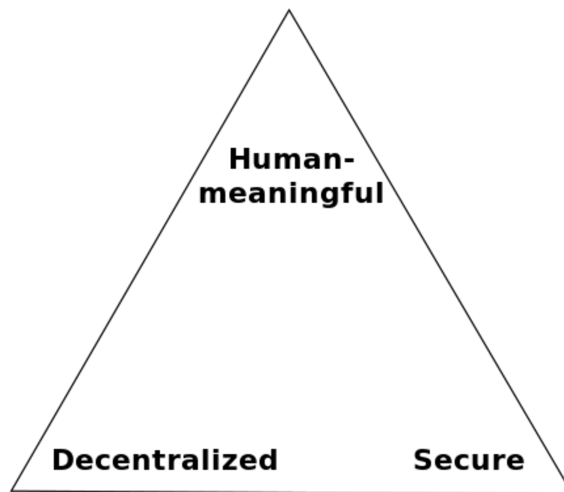

Figure 1.1: Zooko's triangle

Due to the recent actions that were taken by governments to censor DNS, various attacks [9] and following recent security revelations by Edward Snowden, the demand for anonymous access to uncensored information has been growing rapidly each passing day. According to stats taken in 2015, Tor serves a user base averaging two million per day [6]. This growing userbase presses the need for a human-meaningful, decentralized and secure addresses for the onion services.

As described in detail in the literature review, the Onion Name System appears to be at the forefront of creating a proper usable naming system for Tor hidden services. However, the naming system itself possesses the capability to introduce a new attack surface to the onion router. If a naming system is to be implemented to the darknet, extra care has to be taken to make sure that the addition would not impact the secure decentralized nature of the Tor network.

1.2 Research Problem and Research Questions

The research problem is stated as follows.

"What are the mechanisms that could be implemented to enhance the Onion Name System and to enforce better security to the entire system?"

A set of research questions as follows can be generated from this research problem.

- What are the existing vulnerabilities in the Onion Name System?
- What methods could be proposed to overcome these vulnerabilities?
- Does the proposed method perform better than the existing approach in the system?
- Does the proposed method introduce new vulnerabilities to the system?

1.3 Justification for the research

Darknet and hidden services, from the inception, have failed to achieve the widespread success internet achieved. Among several other factors, one of the main reasons for the above, is the usability issues of the darknet. Due to these as well as the added latency, users find it difficult to access hidden services and the surface net alike, using the onion router. The major reason for the usability issues is, as explained above, the lack of human meaningfulness in an onion address used to access a hidden service. As a result, only a limited number of onion services gets frequently accessed, whereas a majority of the onion services, which averages around 45,000 per day, does not get visited. Due to lack of traffic and other issues, a lot of hidden services gets shut down as fast as they pop up. Therefore, it is evident that a proper naming system must be added to the onion router to mitigate the closing down of hidden services, that happens due to the lack of traffic. It can also be speculated that, if the above-mentioned usability issues of darknet get removed, the growth rate of hidden services along with the userbase, will increase.

However, while improving the usability issues of the darknet, it also has to be kept in mind to not to break the existing security and the decentralized nature of the darknet and the onion router. Therefore, any naming system that is to be introduced should be closely monitored to make sure that it does not introduce additional attack surface or give an additional motive for an attacker to attack the network. It also has to be kept in mind that, an adversary can actively or passively monitor network traffic to deanonymize Tor users.

To date, there has never been a reported global adversary attack on Tor, but it has also does not rule out the probability of there ever being one. This statement is made with regard to several observations of the Tor architecture as well as geo-political agreements which are currently prevailing in the world. Tor as a low latency router, was not designed to be secure against a global passive adversary [5]. A global passive adversary, if such an entity could exist, is capable of monitoring the incoming traffic to Tor and outgoing traffic from Tor, without necessarily having the capabilities of decrypting the traffic. Such an entity could easily use traffic correlation techniques to statistically deanonymize users as well as hidden services.

It has to be clearly noted that the threat of a global passive adversary is not as farfetched as it sounds. A country could easily be a global adversary for a user within the country who communicates with a hidden service that is hosted in that country itself. The government generally have the authorization to monitor the traffic and hence the communication between the user and the service could be easily monitored. Due to various geo-political agreements, it also has to be noted that a country may very well be capable of monitoring the network traffics of other countries. As an example, agreements among nations such as the Five Eyes agreement [10] allows powerful countries to share intelligence, which often is signal intelligence. Therefore, it is evident that a powerful player is very much capable of monitoring a significant amount of internet traffic in the world.

However, due to economic reasons, it could be shown that the amount of internet service providers or ISPs that has to be monitored is drastically reduced. This

is because, the number of places where big Tor relays could be hosted is limited. The statement is based on two factors. First factor is that, in order to host a Tor exit node, an ISP with fast connections and cheap bandwidth along with hosting has to be selected. Therefore, it is evident that there is no pressing need to monitor majority of the non-economical ISPs for their traffic in order to deanonymize Tor. In addition to that, a considerable number of ISPs consider the traffic that goes in and out of a Tor relay as a violation of terms of services. Therefore, in order to host a Tor relay an ISP which does not consider a Tor relay, a violation of services has to be picked. These limitations drastically reduced the number of ISPs that has to be monitored in order to truly become a global adversary.

With the above-mentioned limitations and possibilities, if an adversary can even monitor a limited set of ISPs, it has the capability of becoming a global passive adversary. Therefore, special care has to be taken to make sure that introduction of a naming system such as the Onion Name System, does not improve the probabilities of an attack on the Tor network, even in case of a global adversary.

1.4 Methodology

The proposed methodology is of four folds as shown on Figure 1.2. The first step is to analyse the probable attack surfaces on the Onion Name System, to which no heed is paid to typically, such as the possibility of a global adversary. Next step is to evaluate the Onion Name System to find out the alternative mechanisms that could be considered, to change the vulnerable time window of ticket submission in the Onion Name System. The third step of the proposed approach is the replacement of the identified vulnerable components in the OnioNS, through the analysis. The fourth step is to analyse the introduced mechanism to see if it solves the issues mentioned above, as well as to see if it has introduced any new flaws to the onion name system.

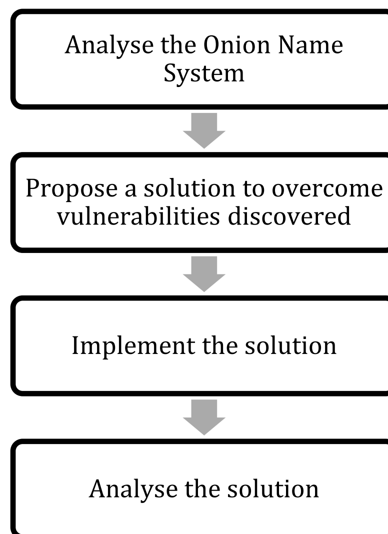

Figure 1.2: Proposed research methodology

1.5 Outline of the Dissertation

The dissertation is structured as follows. The chapter two covers possible and tested approaches to introduce a naming system for darknet and the Onion Name System in detail. Chapter three explores in detail the design of the proposed solution and the implantation details are mentioned in chapter four. The evaluation criteria of the research and the evaluation of results obtained are discussed in detail, in chapter five. Chapter six concludes the dissertation with a discussion about the possible future works for the proposed solution.

1.6 Delimitations of Scope

As a part of the research, suitable modifications would be done to the internal data structures of the Onion Name System in order to evaluate the impact of changes on the system. The content provided by hidden services that requests for names from the Onion Name System will not be considered within the scope of the research.

1.7 Summary

In a summary, this chapter laid the foundation for the dissertation. It introduced the background of the research which led to the research problem, concerned with the security and performance enhancement of the Onion Name System. Additional research questions that arises were then discussed and the need for a research on the topic was justified. The methodology for the research was outlined and the delimitations to the scope were also described.

Chapter 2 - Literature Review

2.1 Introduction

The chapter explores the current approaches and related work on developing a naming system for darknet hidden services. Both the practical approaches that are currently being used as well as more theoretical yet applicable solutions in other domains are discussed in detail in section 2.2. Section 2.3 covers the Onion Name System in detail. In section 2.4, concerns about traffic correlation attacks on Tor and section 2.4 provides a conclusion to the chapter.

2.2 Approaches for Darknet Naming Systems

Multiple attempts and research have taken place to provide memorable addresses to Tor hidden services with the growth of Tor. The most notable of these is the use of Vanity key generators such as Shallot [11]. This brute-force approach is used to find an RSA key which in turn generates a partially-memorable hash value for the address of the onion service. 'facebookcorewwwi.onion' is a hash address of Facebook hidden service, that was generated using similar methods. One major shortcoming of this approach is that it is not capable of computing a full human-readable name across the entire character space within any reasonable timeframe. Also, if the entire character space could be brute forced, the naming system would fail to remain collision-free. To make the case worse for these brute force approaches, Tor plans to increase the size of the character space to be used to identify a hidden service [12]. Therefore, a conclusion could be drawn that, Shallot and other vanity key generators are not capable of providing hidden services with a proper naming system.

A different approach for address manipulation was suggested by Nicolussi [13]. It was proposed to use a dictionary, pre-known to all the parties, to be used to change the encoding of the address from base32 to a delimited series of words. While this solution increases readability of an address, it provides another difficulty. The user still would have to manually enter the address into the browser, making it an impractical solution.

The DNS used in the Clearnet could be considered as an approach to providing human-meaningful domain names to the hidden services. However, the typical architecture of DNS does not give priority to security and therefore, vulnerable to many security threats such as Man-In-The-Middle attacks and DNS cache poisoning. DNSSEC, an alternative that addresses these issues was introduced, but currently, is not widely adopted. One severe flaw in both the DNS and DNSSEC is the hierarchically distributed, yet centralized architecture of the approach. This makes the system vulnerable to government level censorship as discussed in the above section.

GNU Name System [14] also provided a zone-based alternative for DNS. GNS uses a hierarchical zone-based approach which assures the uniqueness of a name within each zone. However, this does not mean the names are globally unique. Therefore, adopting GNS to provide names for Tor hidden services, is doubtful.

Another notable research that was done by creating a system that provides seizure resistant domain names is OnionDNS [3]. This particular research creates a single root server in the Tor network to provide the name service. The service uses, the anonymity guarantees provided by Tor to ensure that the approach is seizure resistant. The method also provides revocation methodologies, if the root server becomes compromised. A major flaw that is seen in the method is that, even if it is difficult, the system could be compromised due to its fundamental centralized approach of having a single root server. In case of a compromised root server or a malicious root server, the recovery process may take time, which makes the system unstable, even for a period.

Namecoin [8] was the first of its kind to achieve all the desirable traits of a naming system while being fully decentralized. It is based on an initial fork of Bitcoin [15] in 2011 and uses blockchains to create an append-only public ledger to hold information about transactions and names. The concept of “miners” that provides a proof-of-work to every transaction that was used provides a mechanism to make sure that the blockchain is irreversible. Although this is a good security measure, the size of an append-only data structure continues to grow with time, creating practical issues of usability of the system. Further, Harry Kalodner et al. [16] analysed the decentralized namespace provided by namecoin and found out that due to fundamental flaws in the algorithm incorporated in distributing domains, the namecoin system was exposed to land rush attacks from its inception. Thus currently, the majority of the namespaces are held by domain squatters, making namecoin essentially a dysfunctional system. Further, the authors have mentioned another flaw in the system is that it does not provide a mechanism to map the one-to-one relationship between the address and the place it points to. However, this particular issue could be addressed by the method proposed by C. Allen et al. in the whitepaper that addresses the creation of a Decentralized Public Key Infrastructure [17]. However, even with the integration of this methodology to namecoin system, it still suffers from usability issues that were mentioned above.

The most promising approach that was proposed for providing human-meaningful domain names for hidden services is the Onion Name System [15]. A detailed description about the OnioNS is given in the following chapter.

2.3 The Onion Name System

Proposed by J. Victors et al. the Onion Name System provides an optional, backward-compatible, decentralized, meaningful and a globally unique, verifiable domain name system for darknet. In order to achieve this, it was proposed to create a decentralized set of Quorum nodes that replicate the set of records associated with domain names for hidden services in a Merkel tree structure. Quorum nodes are a

subset of onion routers in the Tor network, with a predefined set of capabilities. In order to acquire a domain name by a hidden service, a ‘ticket’ has to be generated by the hidden service owner and submitted to the Quorum. From the submitted set of tickets to the Quorum, in order to select a winning set of tickets, a lottery based approach was proposed by the authors. The approach makes the hidden service provide a proof-of-work to generate a request for a particular domain. When generating the proof of work, the RSA private key of the hidden service was required in each CPU iteration, to sign the calculated proof-of-work. This discourages the hidden service owners from outsourcing of the calculation to external parties with higher computational powers. However, the cases where a hidden service owner is willing to share the private key with trusted parties, to generate the proof-of-work is not addressed in this research.

When storing the records or tickets in the Quorum nodes and mirror nodes, two main data structures are used. An AVL tree to store the records which supports efficient retrieval of records for a client query and a Merkle Tree which supports verifiability for a retrieved record. In order to insert new records to the system, as Merkle Tree data structure does not support insertions, the entire data structure has to be regenerated, which takes $O(nlog(n))$ time. Since it is an expensive operation, the tree is regenerated only once, for every 24-hour time period, which happens after 00:00UTC. However, the ticket submission process in the OnioNS imposes some restrictions to the hidden services. The hash commitment of a generated ticket has to be submitted to the Quorum within a limited time window or in a delta amount of time, which is less than ten minutes. This time window also takes place, at a specific static time frame, which is at 00:00UTC. Therefore, to summarise this, the hash commitment of the ticket has to be submitted to the Quorum from 23:55 – 00:00UTC. High-level data flow of the OnioNS is shown in figure 2.1.

Restricting the time interval where a hidden service has to communicate with the Quorum to a specific timeframe which could potentially be targeted by an adversary, raises some questions. A potential adversary can monitor a network during these static time frames, which consumes relatively low amount of resources and

time compared to a 24-hour monitoring station and yield good results as the probability of hidden services communicating at that time interval is very high. Thus, it can be safely assumed that, in case of a global passive adversary or an approximation of such an entity, would be able to deanonymize hidden services, based on this limitation of OnioNS. Therefore, a mechanism to achieve the same set of goals without compromising the secure nature of the Tor network and the anonymity of the users has to be implemented, prior to a widespread implementation of OnioNS.

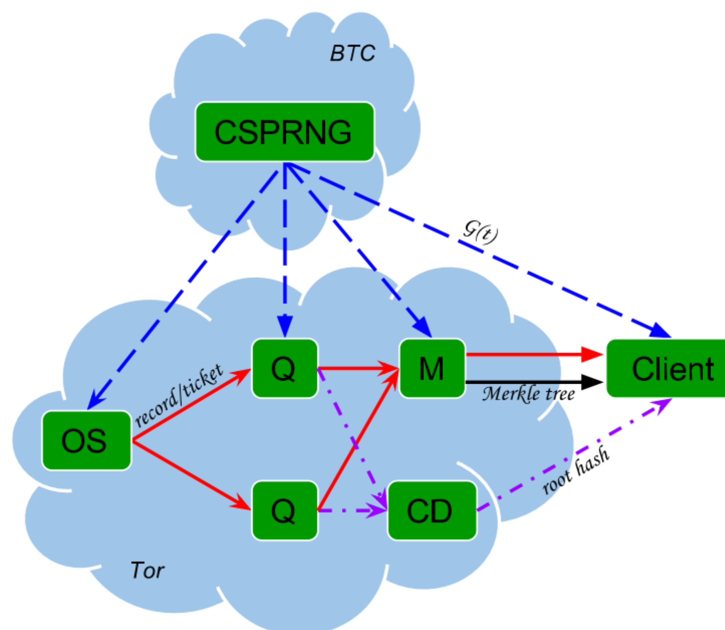

Figure 2.1: High-level data flow diagram in OnioNS [4]

2.4 Traffic Correlation Attacks on Tor

Traffic analysis, confirmation attacks are a special set of attacks conducted by observing the communication patterns between two or more entities in a network. The analysis could be done using several mechanisms, such as by observing the traffic volumes as well as the times the time of communication. Due to the wide variety of ways the messages can be observed, the attacks could be commenced even on encrypted traffic [18].

In a low-latency anonymity network such as Tor, the protection against traffic correlation attacks that could be done by a significant adversary is minimal [5]. In the past few years, there has been a considerable amount of research on the domain. In a research done in 2013, the author states that given enough time, which is roughly three to six months into analyzing the traffic of the Tor network, with a probability of 50% to 80%, a user can be deanonymized [19]. The author K. Müller, in his thesis, also points out, based on the current size of the Tor network, end-to-end traffic confirmation attacks could be successfully carried out.

Therefore, it's abundantly clear that Tor itself is vulnerable, even without a global passive adversary.

2.5 Summary

The chapter discussed the possible naming approaches that could be taken, to provide names for hidden services, along with their advantages and disadvantages. It is evident from the above that the Onion Name System is the most suitable candidate to provide the most practical solution to naming hidden services, based on the current research. However, as mentioned, there are limitations in OnioNS, and these could be vulnerabilities based on the possibilities of traffic correlation attacks as discussed in section 2.4. Therefore, it is abundantly clear that necessary precautions against such limitations has to be taken, in order to mitigate any such risks.

Chapter 3 - Design

3.1 Introduction

This chapter details about the design approach proposed in this research. Section 3.2 covers the high-level research design of the research project and in section 3.3 solutions for limitations of the OnioNS is discussed. The following sections discuss in detail about hash-able data structures, authenticated skip lists and B+ hash trees.

3.2 Research Design

The high-level research design is of five folds. Analysing the current Onion Name System is the first step. This was covered in detail in the literature review. Through the above, the conclusion that OnioNS is prone to traffic correlation attacks were drawn. Secondly, the reasons for introducing the above-mentioned time window, within the OnioNS had to be identified. This identification and analysis of solutions is discussed in detail in the next section. As the third step possible architectural decisions that could be taken in order to mitigate the issue at hand is considered. If the solution is capable of addressing the issues, it has to be implemented as the fourth step. The implementation details of the solution are discussed in detail in chapter four. The fifth step is to analyse the implemented solution. The steps three, four and five is a recursive process, which is to be done over and over again, until a best fit solution to the problem at hand is found.

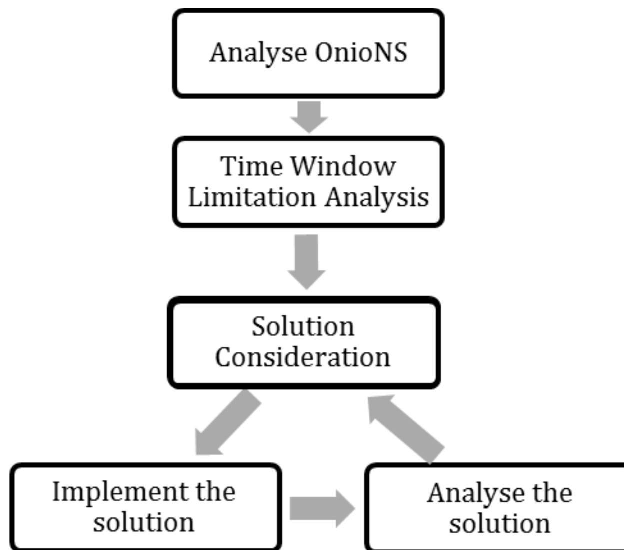

Figure 3.1: High-level design diagram

3.3 Time Window Limitation Analysis

When implementing the Onion Name System, due to design and architectural decisions that were taken, the authors have limited the ticket hash submission window time to a limited static time period, as discussed above. As a way to mitigate the limited time window that appears once every 24-hours, an alternative suggestion would be to replicate the same time window, more often during the same 24-hour period. An example would be to create a five-minute time window for ticket hash submission, every hour. By doing so, an adversary would have to monitor the network 24 more times, to arrive at the same level of results as previously given by the Onion Name System. This implies that the adversary has to expend nearly 24 times resources and time to arrive at the same set of results.

To truly yield the advantage of opening an hourly window, the newly submitted records would also have to be added to the Onion Name System Quorums, in order to make the records readily accessible to a seeker, within a very short amount of time. The current system takes roughly 24-hours for a new name to register a name. Even though the content provided by the hidden services is out of

scope for this research, it has to be noted that these services share extremely sensitive information. When looking at the statistics it can be seen that darknet hidden services often does not last a day or few days at a time, continuously. The reasons for the non-continuity can only be speculated. But it is clear that, due to the above non-continuity, significant amount of information in the darknet gets taken down, prior to be seen by a considerable number of users. On the other hand, if a domain name could be requested and received by a service within a short amount of time, the contents of the service has the chance to be associated by a considerable number of users, prior to being taken down by the service administrator or some other entity that has the physical access to the server.

At the core of the Onion Name System, a Merkle Tree data structure is used in order to provide record verification functionality and to provide proof for the non-availability of a record at the Quorum. Therefore, it is mandatory that each newly accepted record be updated in the Merkle Tree. Since Merkle Trees do not support dynamic updates, for each update, the Tree has to be rebuilt. Rebuilding the Merkle Tree is an expensive operation that takes $O(nlog(n))$ time. In the current version of the OnioNS, this trade-off is rather fine, considering that the tree only gets recompiled once, every 24-hours. If it had to be recomputed every hour, the cost of the re-computation may very well be out of the affordable range for the system. Therefore, it could be seen that a verifiable data structure that is capable of addressing all these concerns, that also provides dynamic updating capabilities would be the better choice in a such a situation. In the following three sections such verifiable data structures are discussed. An alternative data structure must possess the capabilities of verifiability and authenticated denial of existence. Authenticated denial of existence is the ability of the data structure to prove the non-existence of a node within the data structure.

3.4 Hash-able data structures

Any data structure that is capable of being hashed entirely, is a candidate for this solution. After each dynamic update operation on the data structure, the entire data structure can be hashed and the hash value can be stored in a separate location. Verifying the integrity of the data structure is straight forward. A user has to cross reference the hash value to see if the data structure has undergone any changes. If the hash value does not agree with the pre-possessed hash value of the data structure the user can arrive at the conclusion that the data structure has been changed from the last time the user accessed it. However, in order to do this, a tight coupling between the data structure and the hashed value has to be enforced during the implementation of the data structure. Another flaw of the above approach is that even though verification is straight forward, providing the authenticated denial of existence in a data structure with the usage of a single hash value is not feasible. Therefore, the above method is not considered for an implementation within this research.

3.5 Authenticated Skip List

An authenticated skip list is a verifiable data structure based on the skip list data structure. A successful implementation of an authenticated skip list was done by Goodrich et al. [20] and was able to arrive at $O(log(n))$ time complexity for insertion, deletion, update and retrieval of values for the list, in the average case. However, the skip list is a probabilistic data structure which suffers from a worst-case time complexity. In the worst case all the above-mentioned operations take $O(n)$ time complexity to perform. The data structure's capability to land at the worst case with a probability may provide an adversary to exploit the data structure in a way to mitigate the efficiency of the overall OnioNS in the future. Therefore, the above data structure is not considered for implementation in this research.

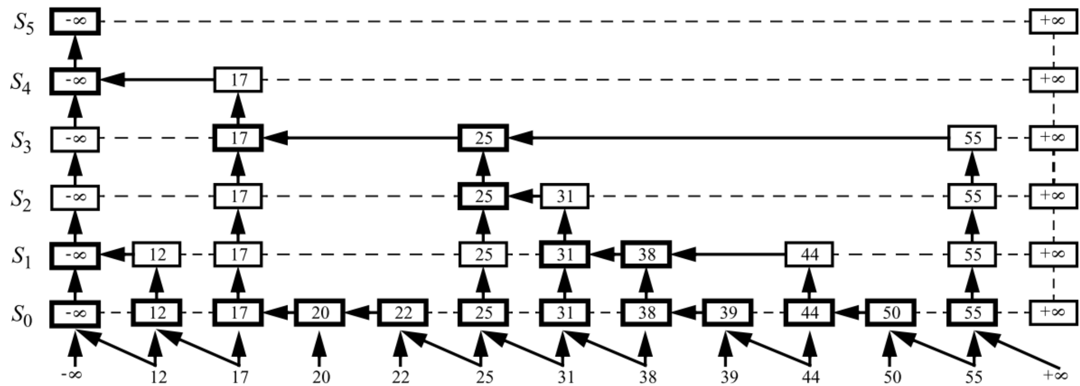

Figure 3.2: Authenticated skip list [21]

3.6 B+ Hash Tree

Since there were no suitable verifiable data structures that were capable of providing all the features provided by a Merkle Tree data structure was not readily available at hand, it was decided to implement a novel verifiable data structure using a B+ Tree. A B+ Tree was chosen as it fulfilled the following design considerations.

1. $O(log(n))$ time complexity at both the average and worst case running times.
2. $O(n)$ storage complexity.
3. Non-leaf nodes only store keys leading to values.
4. Leaf nodes store the entirety of values.
5. Self-balancing.

The importance of a consistent time complexity throughout is self-evident. The importance of all the values being present at the leaf node level is advantageous to provide authenticated denial of existence. Also, the non-existence of a value could be showcased by providing the right sibling and the left sibling of the searched value, and their paths leading up to the root node. In order to do that, all the actual records must be in the same level within the tree, which is, in this case the leaf level, in an ordered manner. The B+ Hash Tree is capable of achieving this property, as it by design pushes all the intermediate values to the leaf nodes.

Figure 3.3 shows the design of the proposed B+ Hash Tree.

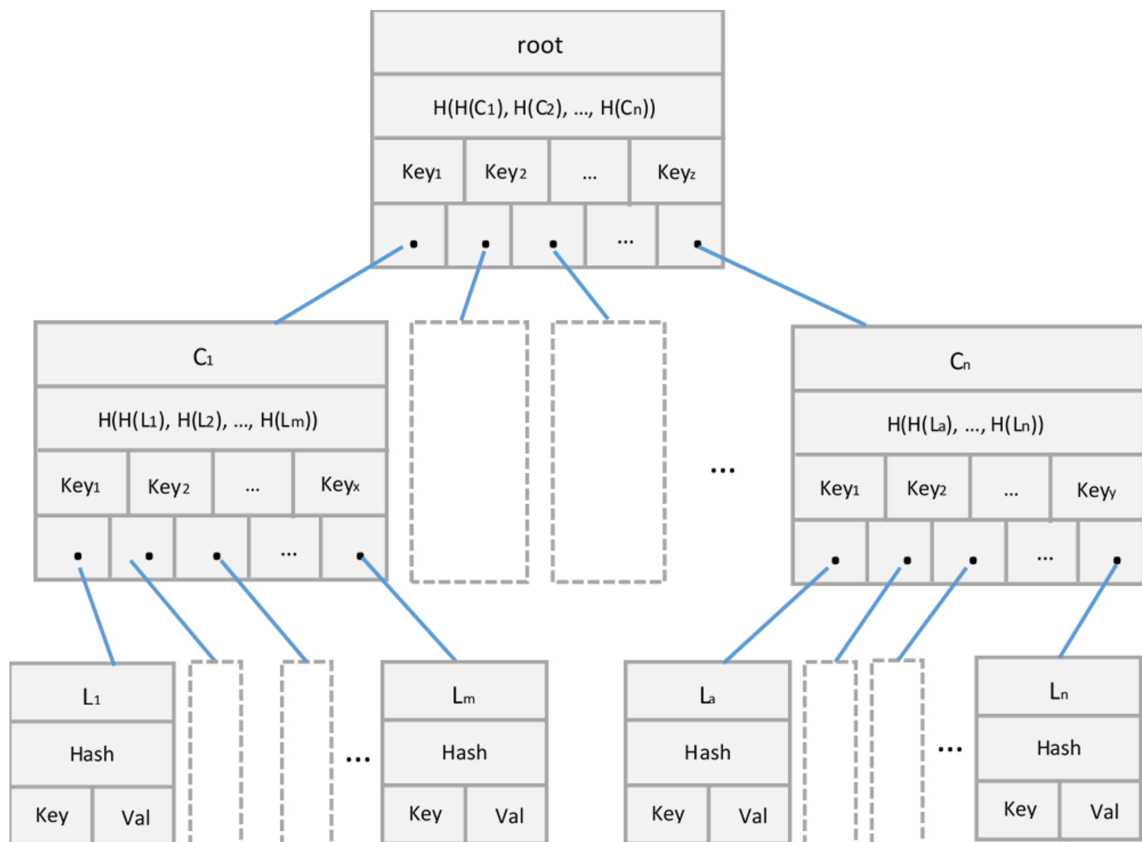

Figure 3.3: B+ hash tree design

The B+ Hash Tree follows the same architecture as that of a regular B+ tree. However, the pointers that are used in the regular B+ tree in order to interconnect the leaf nodes, is not implemented as part of the design as for the current context a single leaf node is capable of storing the entirety of the record within it. Thus, no requirement to further traverse through the leaf nodes is not needed.

In addition to the regular setup of a B+ tree, each leaf node **L** has an additional attribute named Hash, to store the SHA256 hash of the node itself. Each non-leaf node also possesses the same attribute, where the value stored at the Hash changes to the hash of the aggregated hash values off all its children nodes. Therefore, the hash value is propagated up to the root node of the file, where the root node hash reflects the entire tree. For each operation performed on the data structure, the hash value of the changed leaf node, and any other node that were affected by the self-balancing of the tree, has to be recalculated, leading up to the root node. Hence,

every operation performed upon the tree is reflected through the root node hash value.

The B+ Hash Tree algorithm mainly comprises of two parts, namely Insertion and Verification. The algorithmic design of the B+ Hash Tree is given below.

Algorithm 1 Insertion

```
1: target_bucket ← Search for the target bucket to insert
2: if target_bucket is not full then
3:     target_bucket ← Sorted(Keys in target_bucket + key)
4:     Recalculate current hash
5: else
6:     key_list ← Sorted(Keys in target_bucket + new key)
7:     split_value ← Floor((Length of key_list)
8:     target_bucket ← key_list[1...split_value]
9:     Recalculate current_bucket hash
10:    new_bucket ← key_list[(split_value + 1)...END]
11:    Recalculate new_bucket hash
12:    key ← key_list[split_value]
13:    parent_bucket ← target_bucket[parent]
14:    while parent_bucket is not full do
15:        new_key, new_parent ← split_interior(key, parent_bucket)
16:        key ←new_key
17:        parent_bucket ←new_parent
18:    parent_bucket ← Sorted(Keys in parent_bucket + key)
19: Recalculate hash up to root
20: return
```

Figure 3.4: Insertion Algorithm

The above algorithm utilizes 'split_interior' function which is used to facilitate the insertion of interior nodes. The algorithmic implementation of the function is given below.

Algorithm 2 Interior Insertion

```
 1: function SPLIT_INTERIOR(key, parent_bucket)
 2:     all_keys ← Sorted(Keys in parent_bucket + key)
 3:     i ← Floor(length(all_keys + key)/2)
 4:     key ← all_keys[i]
 5:     if parent_bucket = root then
 6:         grand_parent ← new_bucket()
 7:         root[parent] ← grand_parent
 8:         grand_parent[child] ← root
 9:         grand_parent ← root
10:     else
11:         grand_parent ← parent[parent]
12:     new_bucket ← new_bucket()
13:     parent_bucket ← all_keys[1...i-1]
14:     new_bucket ← all_keys[i+1...END]
15:     new_bucket[parent] ← grand_parent
16:     grand_parent[child] ← new_bucket
17:     return key, grand_parent
```

Figure 3.5: Split_Interior Function

By utilizing the above algorithm design, the B+ Hash Tree can be implemented. However, to perform an effective and verifiable search within the data structure for a particular node, further verification algorithm has to be designed. The below algorithm provides a verification mechanism for searched nodes, with the use of hashing.

Algorithm 3 Verification

```
1: root_hash ← Get B+ root hash value
2: current_element ← Search for the element to verify
3: current_hash ← Hash(current_element)
4: while current_element is not root do
5:     sibling_hash ← Obtain hashes of sibling nodes
6:     current_hash ← Hash(current_hash, sibling_hash)
7:     current_element ← Parent node
8: if root_hash = current_hash then return true
9: else return false
```

Figure 3.6: Verification Algorithm

3.7 Summary

A high-level description of the research design and considerations is given in this chapter. The need for a new data structure to replace Merkle Trees was discussed in detail and the preferable features of such a data structure was discussed. Detailed descriptions of various possible data structures were given in the chapter and the novel B+ hash tree data structure was proposed, considering the merits the proposed data structure has over other similar verifiable data structures.

Chapter 4 - Implementation

4.1 Introduction

Software implementation of B+ hash tree is discussed in detail in this chapter. The section 4.2 contains details of the software tools and libraries used for the implementation. Section 4.3 highlights the implementation details of the B+ hash tree on a code level.

4.2 Software Tools

The implementation of the solution was done using Python 3.6. The 'hashlib' python library was used to gain sha256 hashing capabilities for the implemented solution as could be seen in the code level implementation.

4.3 Implementation Details

In order to construct a B+ hash tree as described in detail in chapter three, B+ tree creation approach is taken. The interior and leaf nodes in B+ Hash Tree is constructed with the use of the following 'Node' class. To identify if the node created is a leaf node or an interior node, 'isLeaf' attribute within the class is utilized. The node further has a 'hashValue' attribute, which is populated when the node is added to the Hash Tree. The hash value of the node is calculated at the insertion time. Two other methods, 'before' and 'after' are also defined with the class, which is needed to effectively carry out the search function within the B+ Hash Tree.

```
1.  class Node:
2.
3.      def __init__(self):
4.          self.isLeaf = False;
5.          self.parent = None;
```

```
6.          self.values = []
7.          self.keys = []
8.          self.hashValue = ""
9.
10.     def before(self, key):
11.         for i in range(0, len(self.keys)):
12.             if key == self.keys[i]:
13.                 return self.values[i]
14.
15.     def after(self, key):
16.         for i in range(0, len(self.keys)):
17.             if key == self.keys[i]:
18.                 return self.values[i+1]
```

At the initialization of the B+ Hash Tree, the root node is given and is set as a leaf node initially.

```
1.  class BPlusHashTree:
2.
3.      def __init__(self, root, bucket_size, fill_factor):
4.          root.isLeaf = True
5.          self.root = root
6.          self.bucketSize = bucket_size
7.          assert bucket_size>fill_factor
8.          self.fillFactor = fill_factor
```

Insertions in to B+ hash tree should follow the same algorithm as that of B+ tree. That is, it has to have the logic to handle the three situations of the insertion. The three cases are as follows.

1. Insertion of a new element, when the leaf node has free space.
2. Insertion of a new element, when the leaf node is full, but the immediate parent of the leaf node has free space.
3. Insertion of a new element, when the leaf node is full and the immediate parent node is out of space.

The implementation of the insertion of nodes based on the above three cases is provided in the Appendix A code listings.

During the insertion process, the nodes change its existing leaf nodes, as well as the key values of the interior nodes get changed, and the change has to be propagated up to the root node. Therefore, with each new insertion of a node or a split of an existing node, each node leading up from that point to the root node has to

be rehashed. The changes in the leaf node level must also propagate upwards, by rehashing the node and the entire path up to the root node. This hashing functionality is achieved through the following function.

```
1.  def hasher(self, node):
2.          if node is not None:
3.              while node.parent is not None:
4.                  hashes = ""
5.                  if node.isLeaf:
6.                      node.hashValue = hashlib.sha256(str(node.values).encode('
    utf-8')).hexdigest()
7.                  else:
8.                      for child in node.values:
9.                          hashes += child.hashValue
10.                     node.hashValue = hashlib.sha256(hashes.encode('utf-
    8')).hexdigest()
11.                 node = node.parent
12.             if not node.isLeaf:
13.                 hashes = ""
14.                 for child in node.values:
15.                     hashes += child.hashValue
16.                 node.hashValue = hashlib.sha256(hashes.encode('utf-
    8')).hexdigest()
```

Following searching functionality is also implemented within the B+ Hash Tree to traverse the tree and return the matching results for a given key value.

```
1.  def search(self, node, key):
2.          if node.isLeaf:
3.              return node
4.          elif key < node.keys[0]:
5.              return self.search(node.before(node.keys[0]), key)
6.          elif key > node.keys[-1]:
7.              return self.search(node.after(node.keys[-1]), key)
8.          else:
9.              for i in range(0, len(node.keys)):
10.                 if key < node.keys[i+1]:
11.                     return self.search(node.after(node.keys[i]), key)
```

In order for the B+ hash tree to become a verifiable data structure, a function or a mechanism has to be introduced to check the extracted value against the root hash value of the tree. In order to achieve this, firstly a function must be generated to retrieve all the nodes within the path from a leaf to the root and on top of the results, the verification has to be carried out.

The verification part of the code is implemented in the following manner. The function traces back the entire set of nodes, which are on the path leading up to the

root node as calculated from the above function. At each of these node, the function gathers all the sibling nodes of a given node, at that level. After extracting all the siblings at different levels of the tree for each node, the function calculates the hash values from the leaf node level and checks if it is able to regenerate the parent hash value, each time, traversing upwards in the tree. At the termination point, the function checks if the calculated hash value is equivalent to the root hash value of the tree. If it is, 'True' is returned or else, 'False' is returned. If 'True' is returned, it shows the integrity of the data structure.

4.4 Summary

In this section the details regarding the implementation of the proposed solution was discussed in detail. The important code snippets that are crucial for the successful implementation of the solution was explained. The code snippets that are of relatively high importance and yet could not be inserted in the above section due to space complexities are appended to the appendix A section.

Chapter 5 - Results and Evaluation

5.1 Introduction

This chapter is comprised of the results of the evaluation of the proposed approach against the standard Merkle tree data structure implemented in the python libraries. The results are given out in section 5.2.

5.2 Results

The comparison was carried out between the Merkle Tree data structure implemented in the 'hippiehug' python library [22] against the proposed data structure in the research. In order to do the time analysis for each operation carried out on the data structure, python 'timeit' library is used.

The comparison is done however with the pure Merkle Tree implementation which supports holding python objects at the root level, rather than the actual C++11 implementation that is implemented at the core of the OnioNS. The reasons for the abstract comparison is given in the section 6.4, under limitations.

Both the testing for the running time evaluation was carried out using python 3.5 test environment. The testing was carried out on a computer with an Intel core I7-4710HQ processor running at 2.5GHz using 16GB of RAM, running Microsoft Windows 10 Enterprise edition.

The following table summarizes the time taken by each tree data structure on initialization with different number of starting nodes. Time is given in seconds (s).

Table 5.1: Tree initialization time comparison

Number of Insertions	**Merkle Tree (s)**	**B+ Hash Tree (s)**
1	0.00014492	0.00003038
50	0.00647391	0.00745306
100	0.00490972	0.01413841
200	0.02622285	0.03064444
500	0.03025442	0.05555395
1000	0.07740859	0.08454677
2000	0.18905474	0.16220005
5000	0.43887861	0.3834162
10000	0.97731729	0.81921654
20000	1.99026839	1.68534791
50000	5.35020872	4.511024
100000	11.9428005	9.37703631
500000	-	52.7447996

From the accumulated data, it could be seen that the B+ Hash Tree has performed better than the Merkle Tree when the tree initialization times are compared.

The initialization time analysis is clearly illustrated in the following graph with varying number of starting nodes. The Y axis is given in seconds and X axis represents the number of starting nodes at the initialization time.

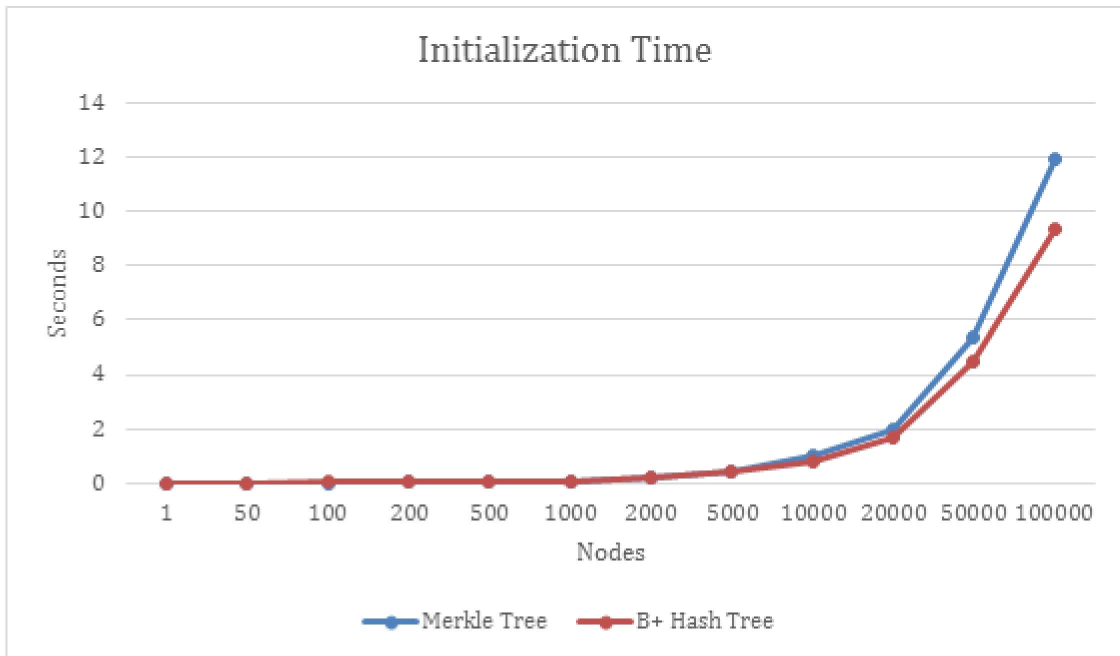

Figure 5.1: Initialization time analysis

In the graph it shows that both data structure implementations draw exponential graphs with respect to the number of nodes to be created at the initialization step. However, it is evident that B+ Hash Tree performs better than the Merkle Tree implementation, when the two graphs are compared.

The next table summarizes the time taken for each new node insertion with for a tree with varying number of starting nodes. The time is given in seconds (s).

Table 5.2: Node insertion time comparison

Number of Nodes	New node insertion time (s)	
	Merkle Tree	B+ Hash Tree
1	0.00007307	0.00001847
50	0.00937114	0.00008662
100	0.01789203	0.00008334
200	0.01572229	0.00009196
500	0.06895379	0.00010017
1000	0.10070835	0.00013712
2000	0.16736268	0.00010879

5000	0.43392372	0.00011536
10000	1.01578839	0.00013506
20000	2.02357564	0.00012644
50000	5.31526174	0.00012357
100000	11.6849509	0.00014081
500000	-	0.0001909

As shown in the table, the two data structures takes drastically different amounts of times to add a new node to an already existing data structure, with a given size.

The following graph illustrates the insertion times taken by each of the two data structures to add a new node to an existing set of nodes. The Y axis is given in seconds and X axis represents the number of existing nodes at the moment of insertion.

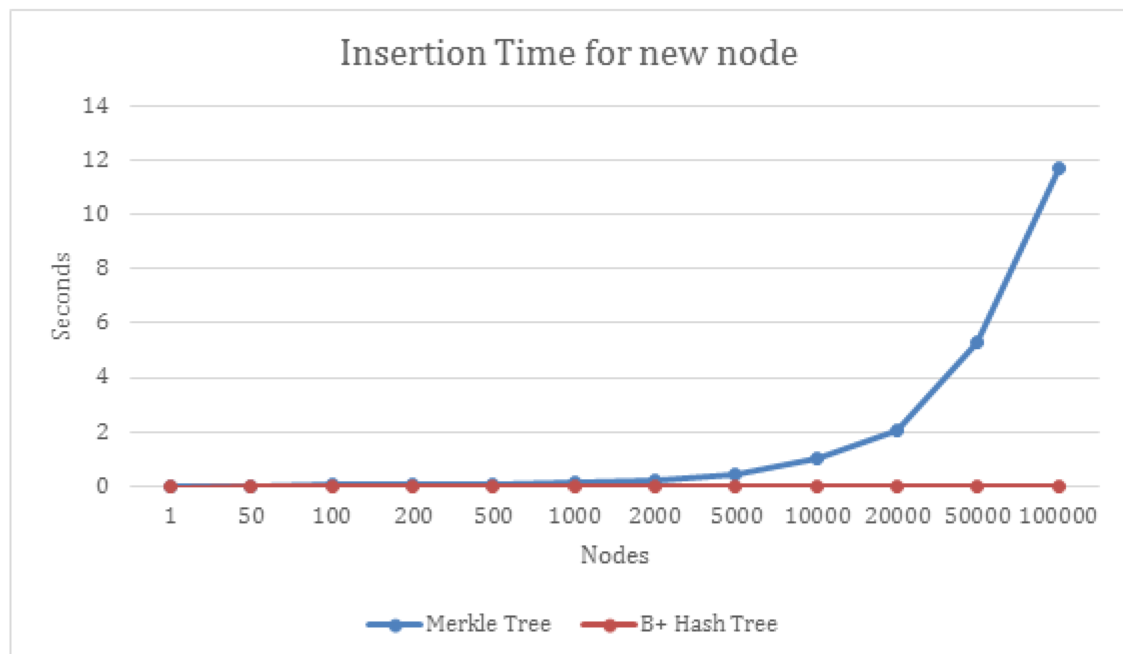

Figure 5.2: Insertion time analysis

The graph shows that the Merkle Tree data structure shows an exponential growth in time taken to add a new node to the data structure. The time taken by the B+ Hash Tree remains linear throughout. The above can be observed because B+ Hash Tree supports dynamic insertions whereas Merkle Tree doesn't. For each new insertion, Merkle Tree recomputes the entire tree, whereas the B+ Hash Tree is

capable of adding the new node to the existing tree, without recomputing the entire tree.

As for the space complexity comparison, B+ Hash Tree uses $O(n)$ space. The Merkle Tree uses the similar amount of space complexity, which is $O(n)$. However, in order to provide efficient retrieval for Merkle Trees, it has to be adjoined with another data structure which supports efficient data retrieval. Within the scope of the OnioNS, an AVL tree is used for this purpose which also has the space complexity of $O(n)$.

5.3 Summary

This chapter elaborated the details of the test setup that was used to analyse both the proposed approach and a Merkle Tree approach to test the capabilities of both data structures holding the same amount of data, undergoing similar operations.

Chapter 6 - Conclusions

6.1 Introduction

This chapter includes an overall review of the research questions, the main research problem, the limitations of the current work and implications for future research works.

6.2 Conclusions about research questions

The analysis carried out of the Onion Name System in fact revealed existing limitations as well as probable vulnerabilities of the system. As pointed out in the justification of the research as well as the literature review, introduction of the OnioNS to Tor network may leave the network increasingly vulnerable to a user or hidden service deanonymization attack done by an adversary. Therefore, it is evident that sufficient countermeasures should be taken in order to mitigate the risk of such an attack.

As a solution to the vulnerabilities a submitting the OnioNS core components to architectural changes were suggested through this research.. After considering various data structures, a novel B+ Hash Tree data structure was proposed in order to provide the same properties as such of a Merkle Tree data structure, which is used at the core of the OnioNS design and implementation.

In the evaluation phase, the novel data structure was compared against the abstract implementation of a Merkle Tree in Python, it was shown that B+ Hash Tree performs better than the Merkle Tree implementation with regard to initializations and efficient insertions. As Merkle Tree has no support for dynamic node insertions, every new insertion is taken up as a new tree initialization and takes a longer time to

accommodate than the B+ Hash Tree. B+ Hash Tree, with dynamical insertion capabilities, performs new node insertions effectively. It is also evident that the Merkle Tree implementation takes up more space as it needs at least two adjoined data structures in order to perform the same operations performed by the B+ Hash Tree. It also has to be stated that even if in the abstract comparison, the B+ Hash Tree performed better, without actually testing it in the OnioNS environment, it cannot be definitively said that B+ Hash Tree implementation performs better at all cases. The reasons for the inability to do the real-world comparison is given in section 6.4 of this chapter.

It was pointed out in the design phase that by creating several limited time windows to submit the ticket hash, within each 24-hour period, the resources invested by an adversary to monitor the network has to be increased by several folds. It was also pointed out that, if it could be provided to submit the ticket hash within the last five minutes of every hour, the resource investment would have to be raised by an adversary to roughly 24 folds. However, by doing so, there is a probability to invent a different vulnerability to the system. The solution may let domain squatters to acquire more number of domains within a given day. At the best case, within a day, with the existing system, a domain squatter could only acquire one domain. On the average and worse case scenarios, depending on the proof of work threshold value, a domain squatter could acquire few more domains within a given day. However, with the new implementation, depending on the amount of limited time windows created within a day, the number of domains a squatter can acquire, in the best case, increases by the number of limited time windows. Therefore, it has to be stated that, even though changing the time window of submission may very well act as deterrent for a passive adversary, it may introduce new loop holes to the OnioNS system. Domain squatters may even be able to perform minimalistic land rush attacks and acquire enough domain names in such a degree, that may even leave the OnioNS partially dysfunctional similar to Namecoin.

6.3 Conclusions about research problem

The Onion Name System certainly imposes security vulnerabilities to the Tor network, if implemented with the current settings. However, there is the capability to change the ticket hash submission interval to minimize some of the identified vulnerabilities within the system. The change could be carried out by implementing a B+ Hash Tree data structure in place of the Merkle Tree data structure, in Onion Name System. As B+ Hash Tree data structure supports dynamic updates, the additional strain that could happen to the system with the additional domain registrations that has to be added with the new submission window, can be handled gracefully. Therefore, based on the research findings, it could be said that B+ Hash Trees is suitable candidate to substitute Merkle Tree data structure within Onion Name System

The research contributed to the domain of security and data structures alike by introducing a novel verifiable data structure. The B+ Hash Tree proposed, is capable of holding values at the leaf nodes and to provide a verification to show the integrity of the data structure through the root node, with a hash tree mechanism. The implemented data structure is also capable of providing an authenticated denial of existence proof to show the non-existence of a node within the data structure. Further, it has to be noted that the B+ Hash Tree may have a variety of use cases in other domains.

6.4 Limitations

The evaluation carried out in section five tested the B+ hash tree implementation against the Merkle Tree implementation of the python 'hippiehug' library. The ideal evaluation to be carried out would have been to test the B+ hash tree implementation against the Merkle Tree implementation in the OnioNS system. However, when building the OnioNS from the code repository at GitHub, it was evident that the current implementation of OnioNS is dysfunctional. As shown in the

figure 6.1, the code failed to build, due to a set of build errors. The code had unimplemented data types, as well as undefined constructor calls which made it evident that the available code was in no running condition. Therefore, the evaluation of the implementation was limited to a context free data structure level, which is not the most fitting evaluation, for the scope of this research.

```
MerkleTree.cpp: In constructor ‘MerkleTree::Leaf::Leaf(const RecordPtr&, const NodePtr&)’:
MerkleTree.cpp:339:60: error: no matching function for call to ‘MerkleTree::Node::Node(const NodePtr&, Botan::SecureVector<unsigned char>)’
     : Node(parent, record->hash()), name_(record->getName())
                                                            ^
MerkleTree.cpp:281:1: note: candidate: MerkleTree::Node::Node(const NodePtr&, const SHA256_HASH&)
 MerkleTree::Node::Node(const NodePtr& parent, const SHA256_HASH& hash)
 ^~~~~~~~~~
MerkleTree.cpp:281:1: note:   no known conversion for argument 2 from ‘Botan::SecureVector<unsigned char>’ to ‘const SHA256_HASH& {aka const std::shared_ptr<std::array<unsigned char, 32ul> >&}’
MerkleTree.cpp:273:1: note: candidate: MerkleTree::Node::Node()
 MerkleTree::Node::Node()
 ^~~~~~~~~~
MerkleTree.cpp:273:1: note:   candidate expects 0 arguments, 2 provided
In file included from MerkleTree.cpp:2:0:
MerkleTree.hpp:23:9: note: candidate: MerkleTree::Node::Node(const MerkleTree::Node&)
  class Node
        ^~~~
MerkleTree.hpp:23:9: note:   candidate expects 1 argument, 2 provided
Record.cpp: In member function ‘std::__cxx11::string Record::computeOnion() const’:
Record.cpp:273:22: error: ‘base32’ has not been declared
  std::string addr = base32::encode(hashStr);
                     ^~~~~~
In file included from ed25519.c:19:0:
./ed25519-donna/ed25519-randombytes.h: In function ‘void ed25519_randombytes_unsafe(void*, size_t)’:
./ed25519-donna/ed25519-randombytes.h:88:26: error: invalid conversion from ‘void*’ to ‘unsigned char*’ [-fpermissive]
  RAND_bytes(p, (int) len);
                         ^
In file included from ./ed25519-donna/ed25519-randombytes.h:83:0,
                 from ed25519.c:19:
/usr/include/openssl/rand.h:101:5: note:   initializing argument 1 of ‘int RAND_bytes(unsigned char*, int)’
 int RAND_bytes(unsigned char *buf, int num);
     ^~~~~~~~~~
```

Figure 6.1: Build errors

6.5 Implications for further research

As expressed in section 6.2, the ticket hash submission time window change may very well introduce additional vulnerabilities to the OnioNS. Therefore, research has to be carried out, in order to identify the proper number of limited time windows that are to be allowed, and the proper time intervals to leave them open, in order to make sure the domain squatters do not get an unnecessary advantage over the system, while still discouraging the traffic monitoring adversaries.

References

[1] L. H. Newman, “How VPNs Work to Protect Privacy, and Which Ones to Use | WIRED,” 2017. [Online]. Available: https://www.wired.com/2017/03/want-use-vpn-protect-privacy-start/. [Accessed: 15-Dec-2017].

[2] D. L. Chaum and D. L., “Untraceable electronic mail, return addresses, and digital pseudonyms,” *Commun. ACM*, vol. 24, no. 2, pp. 84–90, Feb. 1981.

[3] H. Carter and P. Traynor, “OnionDNS : A Seizure-Resistant Top-Level Domain,” 2010.

[4] J. Victors, M. Li, and X. Fu, “The Onion Name System,” *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 1, pp. 21–41, 2017.

[5] R. Dingledine, N. Mathewson, and P. Syverson, “Tor : The Second-Generation Onion Router.”

[6] G. Owen and N. Savage, “The Tor Dark Net,” *Glob. Comm. Internet Gov. Pap. Ser.*, no. 20, p. 9, 2015.

[7] Z. Wilcox-O’Hearn, *Names: Decentralized, Secure, Human-Meaningful: Choose Two*. .

[8] “Namecoin.” [Online]. Available: https://namecoin.org/. [Accessed: 16-Dec-2017].

[9] Ionut Arghire, “Hackers Used Government Servers in DNSMessenger Attacks | SecurityWeek.Com.” [Online]. Available: http://www.securityweek.com/hackers-used-government-servers-dnsmessenger-attacks. [Accessed: 16-Dec-2017].

[10] *Five Eyes*. United States Army Combined Arms Center.

[11] katmagic, “Shallot,” 2012. [Online]. Available: https://github.com/katmagic/Shallot. [Accessed: 17-Dec-2017].

[12] “224-rend-spec-ng.txt\proposals - torspec - Tor’s protocol specifications.” [Online]. Available: https://gitweb.torproject.org/torspec.git/tree/proposals/224-rend-spec-ng.txt. [Accessed: 17-Dec-2017].

[13] M. Gander and S. Nicolussi, “Human-readable Names for Tor Hidden Services,” 2011.

[14] M. Wachs, M. Schanzenbach, and C. Grothoff, “A Censorship-Resistant, Privacy-Enhancing and Fully Decentralized Name System,” pp. 127–142, 2014.

[15] S. Nakamoto, “Bitcoin : A Peer-to-Peer Electronic Cash System,” pp. 1–9.

[16] H. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, “An empirical study of Namecoin and lessons for decentralized namespace design.”

[17] C. Allen *et al.*, “Decentralized Public Key Infrastructure,” 2015.

[18] “Traffic Analysis.” [Online]. Available: https://www.sans.edu/cyber-research/security-laboratory/article/traffic-analysis. [Accessed: 17-Dec-2017].

[19] A. Johnson, R. Jansen, M. Sherr, P. Syverson, and W. Dc, “Users Get Routed : Traffic Correlation on Tor by Realistic Adversaries.”

[20] M. T. Goodrich and R. Tamassia, “Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing.”

[21] “Efficient Authenticated Dictionaries with Skip Lists and Commutative Hashing ∗,” 2001.

[22] “The hippiehug Merkle Tree Library — hippiehug 0.0.1 documentation.” [Online]. Available: http://hippiehug.readthedocs.io/en/latest/#installation. [Accessed: 18-Dec-2017].

Appendix A: Code Listings

A detailed implementation of the B+ Hash Tree, complete with the initialization, node insertion, hashing and searching/verification.

```
1.  class BPlusTree:
2.  
3.      def __init__(self, root, bucket_size, fill_factor):
4.          root.isLeaf = True
5.          self.root = root
6.          self.bucketSize = bucket_size
7.          assert bucket_size>fill_factor
8.          self.fillFactor = fill_factor
9.  
10.     def search(self, node, key):
11.         if node.isLeaf:
12.             return node
13.         elif key < node.keys[0]:
14.             return self.search(node.before(node.keys[0]), key)
15.         elif key > node.keys[-1]:
16.             return self.search(node.after(node.keys[-1]), key)
17.         else:
18.             for i in range(0, len(node.keys)):
19.                 if key < node.keys[i+1]:
20.                     return self.search(node.after(node.keys[i]), key)
21. 
22.     def insert_into_list (self, node, key, value):
23.         for i in range(0, len(node.keys)):
24.             if key < node.keys[i]:
25.                 if node.isLeaf:
26.                     node.keys.insert(i, key)
27.                     node.values.insert(i, value)
28.                 else:
29.                     node.keys.insert(i, key)
30.                     node.values.insert(i + 1, value)
31.                 break
32.             elif i == len(node.keys) - 1:
33.                 if node.isLeaf:
34.                     node.keys.insert(i + 1, key)
35.                     node.values.insert(i + 1, value)
36.                 else:
37.                     node.keys.insert(i + 1, key)
38.                     node.values.insert(i + 2, value)
39.                 break
40. 
41.     def hasher(self, node):
42.         if node is not None:
43.             while node.parent is not None:
44.                 hashes = ""
45.                 if node.isLeaf:
46.                     node.hashValue = hashlib.sha256(str(node.values).encode('
    utf-8')).hexdigest()
47.                 else:
48.                     for child in node.values:
49.                         hashes += child.hashValue
50.                     node.hashValue = hashlib.sha256(hashes.encode('utf-
    8')).hexdigest()
```

```
51.                 node = node.parent
52.             if not node.isLeaf:
53.                 hashes = ""
54.                 for child in node.values:
55.                     hashes += child.hashValue
56.                 node.hashValue = hashlib.sha256(hashes.encode('utf-
    8')).hexdigest()
57.
58.     def insert(self, key, value):
59.         insertion_node = self.search(self.root, key)
60.         node, new_node = self.insert_into_node(insertion_node, key, value)
61.         self.hasher(node)
62.         self.hasher(new_node)
63.
64.     def set_parent(self, node):
65.         for child in node.values:
66.             child.parent = node
67.
68.     def separate_nodes(self, node, new_node, case=0):
69.
70.         temp = node.keys
71.         temp_vals = node.values
72.         if node.isLeaf:
73.             node.keys = temp[0:self.fillFactor]
74.             node.values = temp_vals[0:self.fillFactor]
75.             new_node.keys = temp[self.fillFactor:len(temp)]
76.             new_node.values = temp_vals[self.fillFactor:len(temp_vals)]
77.             new_node.isLeaf = True
78.             if case == 2:
79.                 self.insert_into_list(node.parent, new_node.keys[0], new_node
    )
80.
81.         else:
82.             node.keys = temp[0:self.fillFactor]
83.             node.values = temp_vals[0:self.fillFactor + 1]
84.             new_node.keys = temp[self.fillFactor+1:len(temp)]
85.             new_node.values = temp_vals[self.fillFactor + 1:len(temp_vals)]
86.             self.set_parent(node)
87.             self.set_parent(new_node)
88.             if case == 2:
89.                 self.insert_into_list(node.parent, temp[self.fillFactor], new
    _node)
90.
91.     def insert_into_node(self, node, key, value):
92.         if len(node.keys) < self.bucketSize :
93.             if len(node.keys) != 0:
94.                 self.insert_into_list(node, key, value)
95.             else:
96.                 node.keys.insert(0, key)
97.                 node.values.insert(0, value)
98.
99.             return node, None
100.
101.                elif len(node.keys) >= self.bucketSize and node.parent is not
    None and len(node.parent.keys) < self.bucketSize:
102.                    self.insert_into_list(node, key, value)
103.                    new_node = Node()
104.                    self.separate_nodes(node, new_node, 2)
105.                    new_node.parent = node.parent
106.
107.                    return node, new_node
108.
109.                elif len(node.keys) >= self.bucketSize and (node.parent is Non
    e or len(node.parent.keys) >= self.bucketSize):
110.                    self.insert_into_list(node, key, value)
111.                    new_node = Node()
```

```
112.                    temp_key = node.keys[self.fillFactor]
113.                    self.separate_nodes(node, new_node)
114.                    if node.parent is None:
115.                        new_root = Node()
116.                        new_root.values.append(node)
117.                        if node.isLeaf:
118.                            new_root.keys.append(new_node.keys[0])
119.                        else:
120.                            new_root.keys.append(temp_key)
121.                        new_root.values.append(new_node)
122.                        self.root = new_root
123.                        node.parent = new_root
124.                        new_node.parent = new_root
125.
126.                    else:
127.                        if node.isLeaf:
128.                            self.insert_into_list(node.parent, new_node.keys[0
    ], new_node)
129.                        else:
130.                            self.insert_into_list(node.parent, temp_key, new_n
    ode)
131.                        new_parent = Node()
132.                        temp_keys = node.parent.keys
133.                        temp_values = node.parent.values
134.                        node.parent.keys = temp_keys[0:self.fillFactor]
135.                        node.parent.values = temp_values[0:self.fillFactor + 1
    ]
136.                        new_parent.keys = temp_keys[self.fillFactor + 1:len(te
    mp_keys)]
137.                        new_parent.values = temp_values[self.fillFactor + 1:le
    n(temp_values)]
138.                        if node.parent.parent is None:
139.                            new_root = Node()
140.                            new_root.values.append(node.parent)
141.                            new_root.keys.append(temp_keys[self.fillFactor])
142.                            new_root.values.append(new_parent)
143.                            self.root = new_root
144.                            node.parent.parent = new_root
145.                            new_parent.parent = new_root
146.                            self.set_parent(node.parent)
147.                            self.set_parent(new_parent)
148.                        else:
149.                            new_parent.parent = node.parent.parent
150.                            old_grand_parent = node.parent.parent
151.                            self.set_parent(node.parent)
152.                            self.set_parent(new_parent)
153.                            self.insert_into_node(old_grand_parent, temp_keys[
    self.fillFactor], new_parent)
154.
155.                    return node, new_node
```