Bento hidden services: bringing anonymity to Bento servers

Adarsh Menon

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2025

Abstract

Multiple applications have been made to bring ”non-anonymous” services over to Tor. Bento is one such application, bringing network function visualisation to Tor, allowing individuals to run functions anonymously over the network. But with it’s current design, the capability to share functions is offset by the need to disclose the location of the Bento server. Leaving the server and the hosted functions vulnerable to being taken down. This thesis explores an experimental design which allows Bento servers to be accessed and their functions used by any Bento user, while remaining hidden. Giving users the ability to anonymously use functions on programmable middleboxes, and share them as well in an accessible way. This is possible with a design similar to Tor’s hidden service design, incorporating similar connections, with a detailed directory more suited to descriptions of Bento functions and how to access them. Utilizing the STEM library in Python, we demonstrate a potential approach to implementing this design, while assessing the feasibility of this system.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Adarsh Menon*)

Acknowledgements

I would like to express my deepest gratitude to my supervisor Professor Tariq Elahi, for his irreplaceable guidance, expertise and support throughout the duration of this project. I would also like to thank my Student Advisor Liam McCabe, as well as my family and friends who helped maintain my mental and physical health during this journey.

Table of Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goals . . . 2
- 1.3 Report Structure . . . 2

2 Background **3**
- 2.1 Tor Circuit . . . 3
  - 2.1.1 Introducing Hidden Services . . . 3
  - 2.1.2 Connecting to Hidden Services . . . 4
- 2.2 Bento Box . . . 5
  - 2.2.1 Running functions with Bento . . . 6
  - 2.2.2 Connecting to Bento Servers . . . 7
  - 2.2.3 Bento with Tor . . . 7
- 2.3 STEM library . . . 7

3 Design & Implementation **8**
- 3.1 Function sharing with Bento . . . 8
  - 3.1.1 Vulnerabilities in existing design . . . 8
  - 3.1.2 New Solution design goals . . . 9
- 3.2 Bento Hidden Services . . . 9
  - 3.2.1 Utilizing HTTP Requests for Client API . . . 10
  - 3.2.2 Comparing Websocket connections to HTTP . . . 11
  - 3.2.3 Persistent connection with the Bento server . . . 13
  - 3.2.4 Choosing the right paths . . . 14
- 3.3 Bento Function Directory . . . 16
  - 3.3.1 Server's Registering Hidden Functions . . . 18
  - 3.3.2 Directory and Password management . . . 18
  - 3.3.3 Hashing functions and Bcrypt . . . 19
  - 3.3.4 BFD server and client interaction . . . 21
- 3.4 Token Generation . . . 21
  - 3.4.1 Diving into the uuid package . . . 22
  - 3.4.2 The Pseudo random number generator . . . 23
  - 3.4.3 Alternative approach to handle tokens . . . 24
  - 3.4.4 Verifying tokens . . . 25

4 Testing & Results **26**

4.1 Analysing speed of Connection 26
4.1.1 Amortised Costs 28
4.1.2 Cost from accessing directories 30
4.2 Location revealing functions 31
4.2.1 Issues with packages and anonymity 31
4.2.2 IP revelaing functions 31
4.2.3 Alternative exposing methods 33

5 Evalution **34**
5.1 Anonymity trilemma 34
5.2 Anonymity-compromising functions 35
5.2.1 Utilising Proxies 35
5.2.2 Removing Store requests 36

6 Conclusions **37**
6.1 Future Work 37

Bibliography **39**

Chapter 1

Introduction

”There is more than one way to burn a book. And the world is full of people running about with lit matches.” - Ray Bradbury. This quote is ever more relevant to the internet of today, and the risks of censorship one is open to from deanonymization. Several studies and tools have been made that bring forward non-anonymous services in an anonymous fashion. Valuable additions that allow users to freely explore and express their ideas on the internet without risk of having their access censored by adversaries on the network. Bento[23] is an application that does this, bringing programmable in network middleboxes to Tor. Allowing Bento clients to anonymously run functions on the network. With it's current design, Bento servers are difficult to hide, and the sharing of functions online increases the risk of the function and it's relevant data being compromised. In this thesis we introduce a re-design and additions to Bento, allowing for functions and the servers hosting them to be shared over Tor without the risk of being taken down.

1.1 Motivation

While many new complex and useful internet services are in the hands of users, they typically do not implement them to be used in a private or anonymous manner. Generally, this is due to lack of incentive on the developer's end as it requires more time and effort. Leading to those that consider anonymity in their design, taking longer to release into the market[3][17]. With this, users have also grown accustomed to sharing their data and having less anonymity[29] to use these new services as they're the most popular and viable options. Yet, researchers and developers have steadily been making progress to bring these services to users in an anonymous way. Tor and Bento are such applications that aim to do this.

The motivation for this paper is in a similar vein to the development of those projects. Presenting a modification to Bento, so that it is more resilient to censorship and de-anonymisation attacks, while increasing its usability and shareability amongst its users . Thus giving more freedom and control back to users who wish to protect their data and their ability to use the internet privately, and still having powerful, viable tools that are already available to the open, non-anonymous internet.

1.2 Goals

The aim of this thesis is to present a new design for Bento which brings anonymity to Bento servers. Prioritising anonymity and also the ability to safely share resources, while still maintaining Bento's usability and existing features. We address the anonymity trilemma and analyse the trade-offs involved due to implementing the new features in Bento to accomplish its tasks. The code is meant to only add onto Bento, giving users an option to opt into a more anonymous method of hosting a Bento server. I also introduce a directory for Bento functions, which is called the Bento Function Directory. Allowing servers to share their functions with any Bento user anonymously, just by giving clients access to a function's token. The structure of the directory and it's place in the network will be evaluated under different criteria, and we share our conclusions and other design options that could achieve better performance but were not possible to implement due to limited resources.

1.3 Report Structure

Chapter 2 is the background chapter, where we lay the relevant works and the topics that build the foundation of this project. We provide a brief explanation of what they do and useful tools that will be brought on later. **Chapter 3** describes the new Bento design and the justification for choices made to build it. Within, it explores how the design was implemented and the the modules and libraries added. **Chapter 4** aims to test the new and old design and implementation, their limitations as well as how the new design can be attacked. Then present the results that were obtained. **Chapter 5** explores the results and evaluates it, interpreting and comparing them to the existing and a hypothetical ideal design. **Chapter 6** contains the conclusions made from the new ideas and evaluation brought from the paper.

Chapter 2

Background

This chapter is a brief introduction to the necessary background required to understand the contributions of this paper. It describes the Tor network, its circuits and hidden services. As well as Bento boxes and how it is currently designed to interact with Tor. Finally we explore a part the STEM library that we use within our implementation.

2.1 Tor Circuit

Tor[2], short for ”The Onion Router”, is a non-profit peer to peer network initially developed by the US Naval Research Laboratory, designed to enable anonymous communications online. The Tor network consists of multiple nodes, known as Tor relays. When a client makes a request to a website on the public internet, using Tor, their request would be passed through 3 random onion relays. The 3 relays are known as the guard, middle and exit nodes. They establish a tor circuit as shown in Figure 5.1.

The first relay through which user traffic goes is the guard node. Once a secure connection is established, using a cryptography protocol known as Diffie–Hellman's key exchange [6] between the user and the guard relay, another connection is made from the guard to the middle relay. Once that 2 hop circuit is established, the middle relay establishes a connection to the exit relay; this way the guard does not know the location of the exit relay and vice versa. Finally the exit relay makes the connection out to the public internet and sends the request. The data is returned through this same circuit. The purpose is to make sure that a local adversary cannot link the user to the requested site.

2.1.1 Introducing Hidden Services

These are regular 3 hop Tor circuits, used for browsing the public internet. There also exists websites and servers, known as hidden services, only accessible through the Tor network, which use multiple Tor relays and longer circuits when establishing a connection with a client. They are made with the purpose of increasing resistance to censorship. Allowing any user with access to Tor to host their own website in a more

anonymous fashion and improve unlinkability. However, it can still be safely connected to by other anonymous users on Tor.

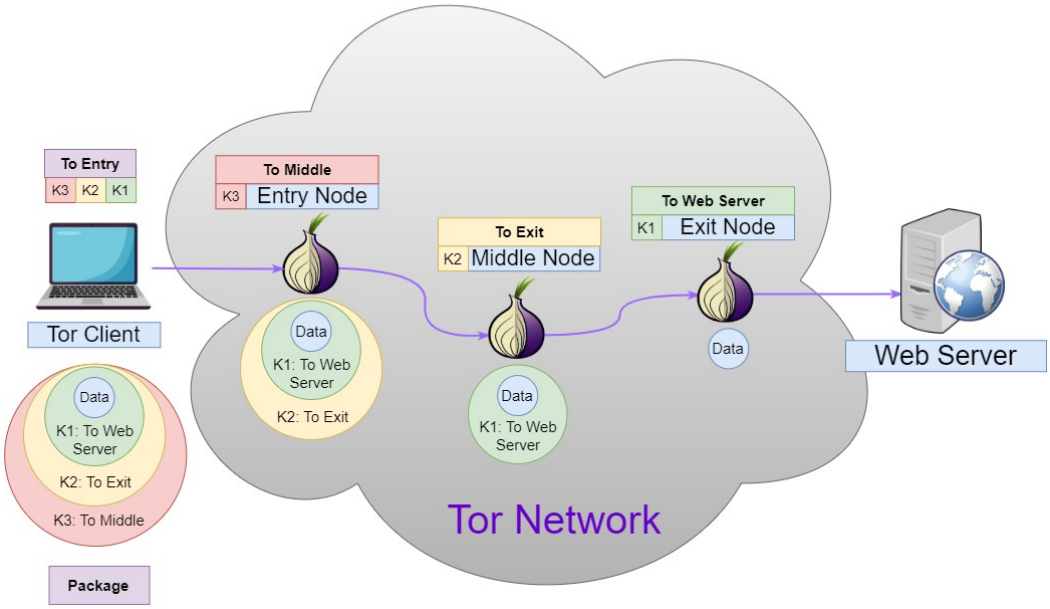


Figure 2.1: A three hop Tor circuit. Data is encrypted in layers, by the public key of each tor node. Then respectively decrypted along the circuit.

2.1.2 Connecting to Hidden Services

Figure(2.2) shows us how a hidden service is shared, and how a user establishes a connection to it. For a server to be set up as a Hidden Service(HS), it selects 3 random Tor relays that will be used by users as initial communication points with the server. The service sends each relay it's public key to set up communications. These are known as introduction points (1). Once selected, the service creates a Hidden Service Descriptor(HSD), which contains the address of the introduction points, a unique onion address made from the public key of the server, and some meta data [11].It then signs the HSD and uploads it to a Distributed Hash table (2) within the Tor network containing all other HSD's, known as the Hidden Service Directory. Now the hidden service has been set up and waits for connections to it.

On the client / user side, they can access the hidden service by obtaining the onion address from a friend or forum on the public internet. They then create a 3 hop circuit to the hidden service directory (3), and use the onion address to find the HSD of the hidden service directory, verified by the encrypted public key in the onion address[12]. The client then sets up a circuit to a random relay. They send the relay a secret key and ask it to behave as a rendezvous point (4), to have extended communications with the hidden service. Using the HSD , they then connect to one of the introduction points for the given onion address. The client then sends an introduction request [21] to the

relay which is passed to the hidden service (5). The request contains the location of the rendezvous point and a secret key to verify their identity at the tor relay.

Once the hidden service receives the introduction request, it creates another 3 hop circuit to the rendezvous point, and verifies themselves using the secret key (6). The rendezvous point then connects the two circuits, from the client and the hidden service, establishing secure communications between them.

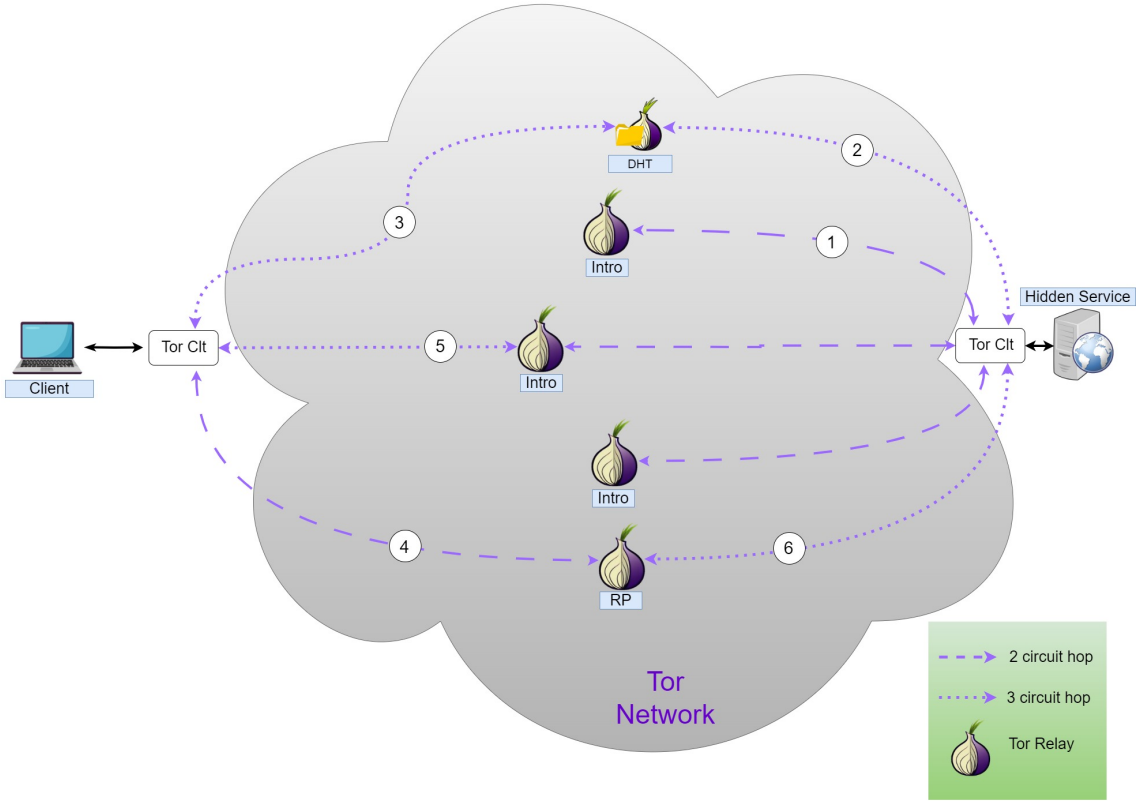


Figure 2.2: Example network overview of a hidden service connection

In exchange for increased latency we get increased anonymity , hiding the location of both the user and hidden service while still establishing communications. We explain hidden services in detail as it is the basis of our design of hidden functions for Bento to achieve similar levels of anonymity and shareability.

2.2 Bento Box

Bento[23] is a service designed to work on top of tor, introducing programmable in network middle-boxes in an anonymous fashion.It attempts to bring this property of Network Function Virtualisation to Tor, while preserving its anonymity features and goals. Bento servers were designed such that clients can offload processes on other machines over tor without modifying Tor itself. The processes themselves are user-written and provided functions that can be ran on machines running a Bento Server.With

no constraints on the function code itself, but restricting the resources and libraries allocated to it set by the server host machine.

2.2.1 Running functions with Bento

When a Bento server is ran, it spawns and dedicates a container for each client's function sent to it. Using the I.P. address, the client connects to the Server. For a client to use any particular function with an active server, the process is as such:

1. **Store Request** : Client sends a request to the server with the function name and code (see Figure 2.3). The server generates a token and saves the details sent in a .txt file. Returning that token to the client.
2. **Execute Request** : The client writes the call to the function, with any arguments, and sends an execute request containing the call and the right token. The serve then attempts to execute the request on its machine in a conclave, returning a session ID if successful and an error message otherwise.
3. **Open Request** : If the function returns data using Bento's "api.send()" call, the client can send an open request after executing the request. This allows the client to receive data produced by the function, from the server.
4. **Close Request** : If the function is still running and the client would like to stop exchanging data with the function. They can send a close request with their session ID, letting the server know that it can terminate the function and reclaim its resources.

Clients can use a combination of these to freely make use of functions, anonymously. Once the function terminates or the client disconnects from the server, the server reclaims the resources taken by the function during runtime. It maintains the token, function name and code in storage within the Bento machine.

```
function_name= "browse"
function_code= """
import requests
import zlib
import os

def browse(url, padding):
    body= requests.get(url, timeout=1).content
    compressed= zlib.compress(body)
    final= compressed
    if padding - len(final) > 0:
        final= final + (os.urandom(padding - len(final)))
    else:
        final= final + (os.urandom((len(final) + padding) % padding))
    api.send(final)

"""
DEBUG:  Parsing store request
DEBUG:  Parsing execute request for token: 08de2755-f144-4547-9eb4-f14de113817f
DEBUG:  Parsing open request for instance: 681d185b-7e20-454b-9989-23e983a45ce1
```

Figure 2.3: Example of function code "Browser" sent to the server by the client. Server produces a token on successful execution request and session ID on open request

2.2.2 Connecting to Bento Servers

Bento servers are currently designed to communicate with Bento clients by using socket connections. A socket is one endpoint which a server can use to listen for connections , allowing for client application to connect with it. Creating a two way communication link where data can be sent back and forth between the client and server for long periods of time. Socket connections use the TCP/IP protocol to ensure the delivery of bytes being sent back and forth.

Specifically, Bento servers use the `socket` module in Python. Listening for the requests made by Bento clients who also use the same module to send those requests. To form a connection Bento servers need to share their IP address and port number to allow for these connections, which necessitates Bento servers being non-anonymous to use publicly. If connected directly, Bento clients also leak their IP, but sockets connections can be made over Tor allowing for it to be masked.

2.2.3 Bento with Tor

If a Bento client is running the Tor client alongside it. They can use the torify module from the Tor package to send function requests and receive data from the Bento server through the Tor network via a three-hop circuit. If a Bento machine also runs a Tor client (or is itself a Tor relay), it is can connect to the Tor instance as well.

Using the STEM[1] library, users can then write and run functions on the Bento server to use services provided by Tor. This functionality is brought safely through setting a STEM firewall which functions must connect to refer it's routines. The firewall will then handle which function control a given circuit, as well as limit the routines available to them as set by the server's policy.

The user still needs to know the location of the Bento machine to communicate with it's Bento server in the current design. Leaving the machine de-anonymised. We further make use of Bento's integration with Tor and the STEM library to change this.

2.3 STEM library

STEM[1] is an open-source controller library for Tor, on Python. Allowing us to use Tor's protocol to script against the Tor Process and monitor Tor instances using a convenient interface. Using this library, developers can start Tor clients, create hidden services, and communicate with the Tor instance.

The STEM library has five core modules that can be used. Three of which are called, for this redesign. The **control** module, the module used to interact with the Tor control socket and the controller used to talk with Tor. The **process** module which contains helper functions for working with tor as a process. And finally, the **connection** module for connecting to the tor instance.

Chapter 3

Design & Implementation

In this chapter, a new design for Bento is introduced and explored. Adding another layer of anonymity and showcasing a way to share functions amongst anonymous users. We explain the issues with Bento as it exists, to use it for this purpose, and our design and implementation approach to address them.

3.1 Function sharing with Bento

Bento in its current design, has a system implemented that allows users to share functions that they have stored in an active Bento server. This can be done with a token that is generated by a server when a valid store request is received from a Bento client, with the function name and code. This token is then sent back to the client on a successful store request to be used as a key to access the stored function at a later time.

3.1.1 Vulnerabilities in existing design

Figure 3.1 shows how a client stored function can be accessed by another user with Bento now. Both the Function creator and user remains anonymous to the Bento server as they are connected using a short, 3 hop tor circuit. The Bento server itself is not hidden. The user has to know the IPV4[14] address of the server, which is its locating address. For a function to be shared with multiple anonymous users, it will have to share the IP address of the server that stores it along with the associated token. If an attacker receives this information, they can easily perform a Denial of Service attack[15]. Which compromises the server's ability to share their functions. Servers publicly sharing their I.P address can also be blocked by another content provider or internet service provider (ISP), to prevent attempts at anonymity. This is already done with connections from many Tor relays ([28]). Ultimately, hindering access to these Bento machines and the functions hosted within them.

3.1.2 New Solution design goals

Due to the sharing of their iP addresses, Bento servers are at increased risk of censorship as more users begin to use and share them.The solution presented is a redesign that allows Bento servers to hide their true location. The choices attempt not only to hide the server, but adhere to the following policies:

1. Bento Servers can opt in to becoming anonymous or not.
2. New design preserves server Anonymity as well as functionality.
3. Network Latency in the new Bento design is minimised.

I aim to maintain these policies and Bento's original principles throughout this design. As to accomplish the goals set out in the beginning of this thesis while not losing focus as to the purpose and use of Bento.

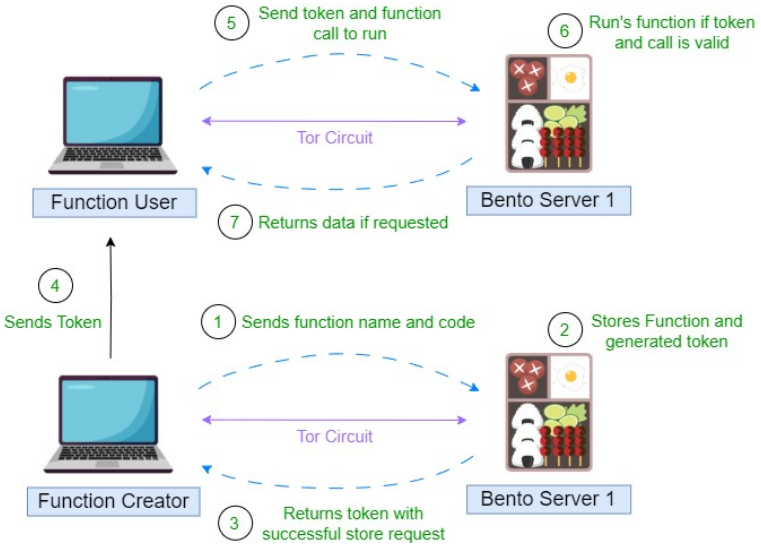


Figure 3.1: Example of how a stored function can be accessed and used by another client

3.2 Bento Hidden Services

This project introduces Tor's hidden services to Bento. As hidden services allow a web server to set up communications indirectly with anonymous Tor users, keeping the server anonymous. We use this tool to allow users to indirectly access our Bento servers as well, introducing anonymity on the server side of Bento. Bento was designed to be used only within Tor[23] and we further enhance it' privacy using the same network

Using the STEM[1] library, we can give Bento servers the ability to start a Tor client and a hidden service within a new python file, which can be run optionally. This hidden service runs simultaneously alongside the Bento server. Both listening on separate ports on the machine and relaying information to one another. We use a web framework known as Flask, to host the hidden service using python and handle communication and requests to the onion url[26] .Once a hidden service is active, it will have a .onion

domain name that can be used by anonymous Tor users, to send and receive requests through their tor client. But users will have to communicate with Hidden services using HTTP requests[13], which work differently to the TCP socket connections(See figure 3.5) that Bento's original design had used, allowing the user and server to communicate with each other.

The goal is to allow users to perform the same actions on Bento while preserving anonymity on the server side. So in the new design, we use a combination of TCP socket connections and HTTP requests to allow Bento clients to communicate indirectly with Bento servers, through hidden services.

3.2.1 Utilizing HTTP Requests for Client API

I designed the hidden service such that it will have multiple routes the user can access with it's onion address. These routes are accessed by clients by altering the "path" of the URL. In a regular hidden service, they would be HTML webpages that a Tor browser can receive using the Hypertext Transfer Protocol, also known as HTTP, and sending an **HTTP GET-request** to the hidden service via a TCP connection, using the URL with the specific path (See Figure 3.2). On the internet, this type of request is the most common and the primary mechanism for information exchange[24].

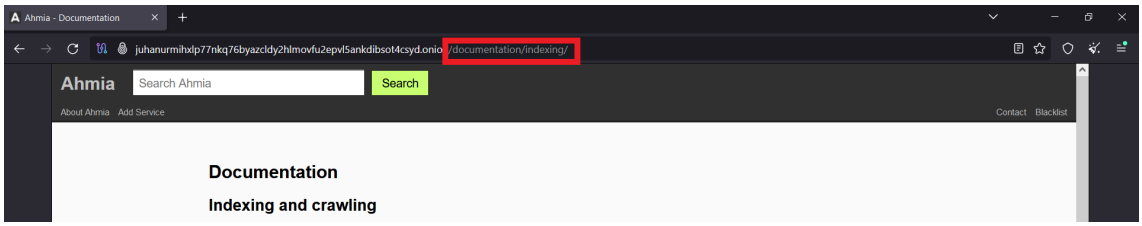


Figure 3.2: Highlighted path of an example onion address

HTTP GET-requests allow users to receive a response from a server, but does not allow users to send data. For our purposes, the server needs to receive unique client data and utilise multiple paths. Each path containing a function made for Bento clients to relay requests to the Bento server through the hidden service. Previous Bento client functions such as *Store-request* and *Execute-Request* are now accessible in a hidden way through sending requests to their respective paths "address.onion/store" and "address.onion/exec". These paths would run separate functions with the arguments provided by the client . The server is able to receive this using POST-requests.

HTTP POST-requests are a less common method of client interaction with URLs, that allow users to ask a web server to accept the body of the message sent to them within their request [25]. I developed a Client API which exclusively uses POST requests to communicate with the Bento server. The POST requests would be handled by these different paths, while the main onion address with the empty path is used as an information page to explain the purpose of the different available paths, which can be obtained via a GET-Request. Within the POST-request, clients place their arguments in JSON-format, as then the data does not have to be de-serialised at the server and can

maintain an expected format when received by the hidden service. The hidden service validates if the information is appropriate for the function and reformats it to send to the Bento server.

The largest limitation here is users cannot form a continuous bi-directional connection like they previously could with Bento. We tackle the idea of using socket connections with hidden services and compare them against using HTTP requests. We also attempt to compensate for this issue by having the hidden service act as a client and establish socket connections when communicating with the Bento server.

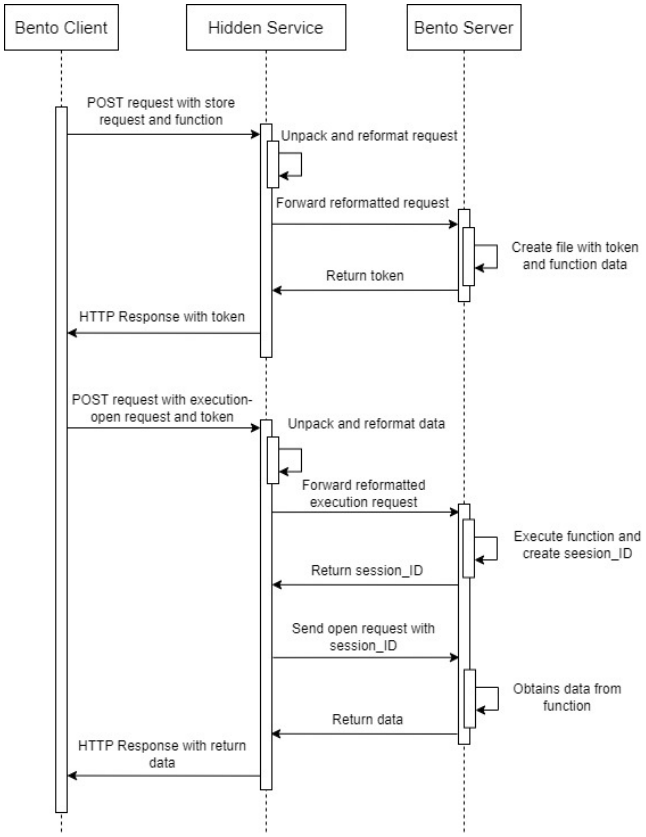


Figure 3.3: Sequence diagram of user storing, executing and receiving data from a function

3.2.2 Comparing Websocket connections to HTTP

In setting up an HTTP-request-response model to form communications between the Bento client and the hidden service, we see that the client is required to send a new request for every server response. This conflicts with how Bento was originally designed, as it is indented to have a persistent connection with the client.

To solve for this problem was challenging, initially I decided that it may be better to not use HTTP to connect to the hidden service and considered using a WebSocket connection instead. This provides a bidirectional communication channel just by initiating a single HTTP request. Enabling a persistent connection with the Hidden

service and allowing for better integration with Bento in theory. Yet the benefits came with some extra costs in practice which I compared against HTTP :

1. **Communication model**:
    - HTTP connections last from the moment a request is sent to the server, up until a response is returned. Then the connection ends, and for our purposes, the user will have to send another request to the hidden service so the server can push more data back to the client.
    - Websockets keep the connection persistent between the client and the server. It is stateless. The communication system is bidirectional and similar to the existing socket connection expected to be used through the tor circuit between a client and a non-anonymous Bento server.
2. **Integration**:
    - HTTP adds extra steps for a Bento user to communicate and do the same tasks Bento previously did. Some functionality requiring persistent back and forth is not possible due to the need for repeated requests and the connection being stateless.
    - Using Websockets would be similar to how Bento handled connections originally. But the connection must be set through the Hidden service which is a web server, which is a very uncommon way of communicating through the web in general [18], much less Tor. Bring about more complexity into the design using hidden services.
3. **Latency Overhead**
    - HTTP request has worse overhead due to to tearing down the connection between client and serve each request-response cycle. Websockets do not do this,and so have less overhead once a connection is formed. But forming a connection with socket.io leads to frequent timeouts that can be sporadic. Creating drastic differences in the execution of any one function.
4. **Anonymity & Security**
    - Websockets introduce privacy concerns that HTTP is less affected by. Due to it's bidirectionality, it is harder to control what data is being transmitted by either server or client side [8], to make sure anonymity is not broken. The persistent connection through the longer circuit can also be easier to monitor by an adversary, due to its uniqueness within the Tor network.
    - With HTTP, the user controls the pace of the requests, and the server can only send data upon each request. The user and the server can decide what sort of data can be sent back and forth more rigorously.

Ultimately, I chose to use only HTTP requests as my method of communicating with the hidden service. The deciding factors were the increase in complexity and the privacy risks that come with setting up a websocket. As well as the lack of documentation to form sockets consistently, as modules such as `Flask-SocketIO` and `websockets`

consistently timeout when connecting to hidden services, with no obvious work around. With better documentation around using websockets over hidden services, future work could integrate websockets into this system instead of HTTP, while preserving the system's anonymity.

For now , we attempt to maximise this form of communication and find unique ways to utilise the HTTP form of communication, to perform different tasks with Bento while adhering to our anonymity goals.

3.2.3 Persistent connection with the Bento server

```
app = Flask(__name__)

@app.route('/open', methods=['POST'])
def exec_open_req():

    sock_hs = ClientConnection('127.0.0.1', 8888)
    data=request.get_data() #data received from client
    jsonified = json.loads(data.decode('utf-8'))

    call = jsonified['1']
    token = jsonified['0']

    session_id, errmsg= sock_hs.send_execute_request(call, token)
    sock_hs.send_open_request(session_id)

    data, msg_type= sock_hs.recv_output()
    response = flask.make_response(bytes(data))

    return response, 200

app.run(host='127.0.0.1', port=5000)
```

Figure 3.4: Example code snippet of hidden service handling an execute and open request for a given function on behalf of the client

Once a connection is formed between the client and the hidden web server. The web server would communicate with the Bento server similar to how a user did with Bento originally. After establishing a 6 hop tor circuit (see Section 2.1.2) with a client and successful reading a POST request, the server will use the existing Bento client API and form a TCP socket connection with the Bento server for bidirectional communication.

Figure 3.5 shows us how the python socket library is used to connect to the server and exchange data. When the Bento server is set up, it is always listening for a connection on a port, but only connections within its local network. Accepting connections requests from the hidden service on the same machine, but no longer allowing direct external

network connections, as this is only possible by a client if given the server's I.P address which is intended to stay hidden.

See Figure 3.4 which represents an example hidden service with one path. `@app.route` is used to create path, in this case the '/open' path . We also specify that this path accepts HTTP-POST requests within 'methods'. Within the path we specify the function to run, The Flask server uses the `ClientConnection` class Bento designed, to establish a socket connection with a Bento server within it's local network. This would be true for any path wishing to communicate with the Bento server.

This function in particular looks within the body of the POST request, which should contain a call and a token. They are both used to execute a function for the client and then receive its first output as `data`. That is then made into a response.If the task is successful, the hidden service closes the connection with the Bento server and returns the response to the client before their connection is torn down.

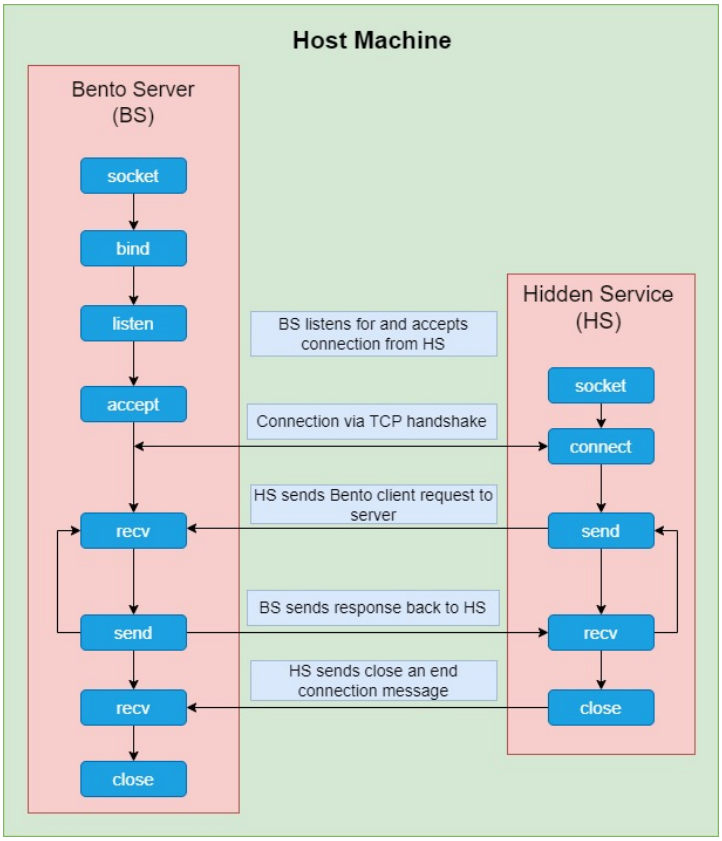


Figure 3.5: TCP socket connection between Bento server and Hidden service, using the python socket library. Source [19]

3.2.4 Choosing the right paths

The general design of a Bento hidden service, is to have 4 paths that a user can send requests to. The first 4 rows of Table 3.1 show the implemented paths and explain the purpose of each. /exec and /open do not capture all types of functions. This is where we tackle the limitations of HTTP connections and how we adapt to them.

Path	Purpose
/store	Used to store functions and returns the respective token
/exec	Executes function associated with token and returns success upon completion
/open	Executes function associated with token and returns data produced by function
/register	Registers function into the Bento Function Directory

Table 3.1: Paths implemented in the current design of Bento Hidden Services.

As mentioned before in section 3.2.2, there are limitations with our system due to using HTTP as our form of communication with the hidden service. Having a generic path made to execute a function associated with a token, and receiving it's return data once, like the example shown in Figure 3.4 does not capture how some functions work in Bento.

Take a website tracking function for example. This function periodically looks up the number of visits/hits a website gets. It returns this data periodically as well, through the `api.send` call, which is received by using the `recv_output()` method shown in the above snippet. If the above path is used to execute this function, upon receiving the first message, the service will send the response immediately to the client and terminate the connection with the Bento server, which in turn terminates the ongoing function.

This is where different, custom paths can be used to capture instances such as these. It is an extra process, but one that allows for unique functions to be executed as they were intended. The client may not be able to receive the data periodically, but the data can be saved in a list and sent back as a response after a given amount of time. An additional note is that these paths are added by the server. They can restrict or allow the types of functions that can be ran through this hidden service system. You can even write specific code that users can choose to run by visiting a custom path.

They can also choose to remove the option for clients to store requests on their server, only allowing administrators who have access to the bento server to store tokens and requests. But for server that do wish to have that function, it will accumulate a large number of tokens. To make these easier to search and use, we also introduce the Bento Function directory which is made as a way to list publicly registered function to be uploaded by servers and be used by clients while both remain anonymous.

3.3 Bento Function Directory

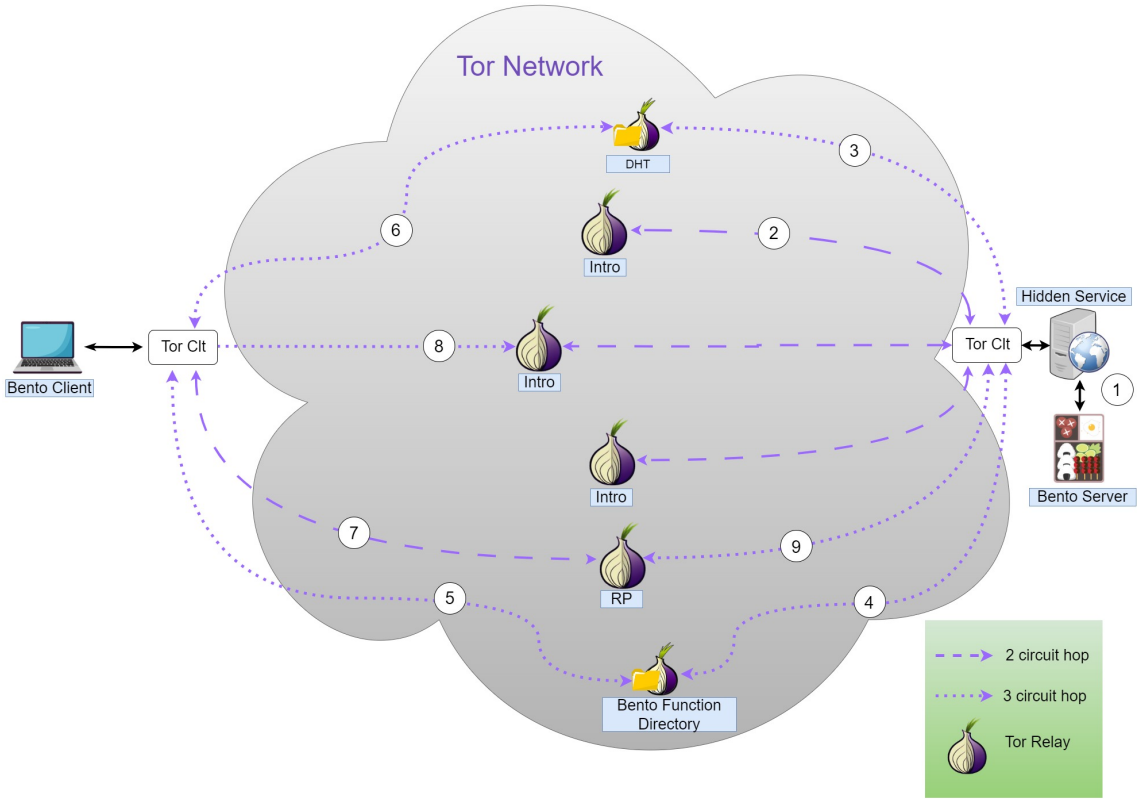


Figure 3.6: Entire layout of connections from client to hidden Bento server

The procedure required for a Bento client to connect to a hidden service is similar to the process shown in Figure 2.2. The user would need to know the onion address of the service to connect to it. Additionally, to use an existing Bento function, a user would also need to know its token to send to that Bento server. Thus the client would have to know both the token of the function and the correct onion address to access the desired function.

A security and usability issue arises due to the client needing both values to connect to the right function. A server could have its onion domain compromised, or be temporarily shut down and lose access to its original domain when rebooted. Figure 3.7 shows an example attack when exploiting this system. (1) The desired server hosts `puppy.py` within Token_X.JSON and shares this information publicly via the clearnet. They then have their private key to generate the onion domain (as described in 2.1.2) compromised and decide to change their key and generate a new url ”safe.onion”. (2) User is unaware of the change and wishes to use the `puppy.py` function and receive some puppy pictures. (3) A malicious server uses the compromised private key to launch a hidden service using the old domain ”compromised.onion”. They also host a Bento server which contains the same Token_X.JSON which runs and returns malicious data when called.

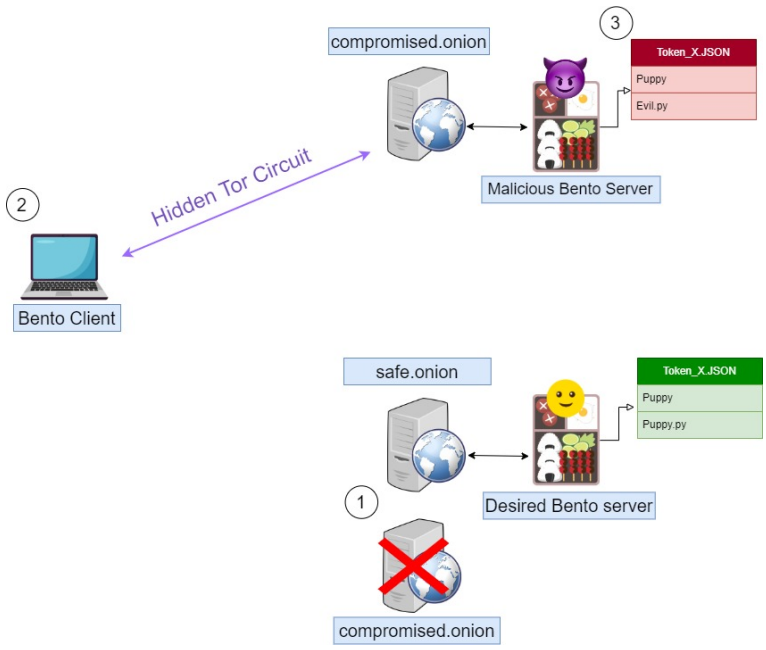


Figure 3.7: Example of a Bento client connecting to the wrong domain and running the wrong function

The Bento client is now under risk of receiving and running a function they did not intend to run. The desired Bento server will then have to share its new domain for clients to be able to access the functions with the same token. More so, any given token for a function x can be copied by another malicious Bento server linking to a different function y. A client who intended to use function x in one server could mistakenly be tricked into using function y if they are not connecting to the correct onion domain.

This problem is tackled by the Bento Function Directory (BFD). Which is a directory, that is set up on a Tor relay to hold all publicly listed tokens. The purpose of the directory is so anonymous clients only need the token to access a shared function, without needing to look for the onion domain as well. Furthermore, every token is unique within the directory. Any function registered into the directory with a given token will remain associated with that same token.

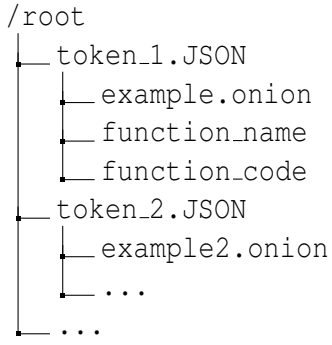
The Bento function directory is set on a Tor relay, managed by a server controlling access to it which we refer to as the BFD server. It is only accessible through the Tor proxy. Figure 3.6 shows the additional connections now required when a user attempts to access a function for the first time. Steps 4 and 6 in the above figure are the additional steps taken. Step 4 is where the server uploads a unique token, it's respective function details, it's onion address at the time, and a unique password to the directory . In step 6, the client receives that token and its onion address, where it can use the newly implemented Bento API to make a connection and run the associated function.

Besides allowing users to only need the token to run a function, the directory is intended to act as a hub for all publicly listed functions. Such that is similar to how Bento previously worked. Where each stored function can be ran just by having access to the server and knowing it's token.

3.3.1 Server's Registering Hidden Functions

In the current system , a function can be registered once it is stored in the Bento box and assigned a token. By implementing a new function in Bento, a server can directly upload its token to the directory by using a 3 hop circuit to connect to it. But we can also give clients the ability to register functions themselves, by adding an optional path as shown in Table 3.1 and upload the token from the Bento Server on their behalf. Of course, Bento servers can take away this option if they wish to allow only administrators of the server to register these functions. Or if they wish to not share tokens publicly and only to trusted users, they can still share the token and onion address separately, without using the directory at all.

Within the directory, each token uploaded is saved as a JSON file. Along with the given token, the server provides the details for each token as listed below:



The onion domain is required for the connection. But the function name and code associated with the token, are also listed along with it to help users understand what the code is and verify the token they have received is correct.

The directory simplifies and ensures that each token uploaded in the directory is unique. A challenge encountered is that once a token is associated with an onion service, it will remain associated with it. If a server wishes to change its onion address, due to it's private key being compromised[31] or they simply lost the private key required to generate the onion address, the token can no longer be used by clients to connect to the server hosting the function. Servers will have to regenerate tokens for functions every single time they decide to change their onion address, and upload those details to share them each time.

To tackle the issue, we want to allow servers to alter the directory, but only for the tokens they upload. So in addition to providing the token details, the server provides a password for the given token when registering it. Section 3.3.2 discusses the design of the password system and the tools used for encryption.

3.3.2 Directory and Password management

On creation, a Bento server can now receive a password as an argument. This password is expected to be provided by the user and saved securely elsewhere, but is also randomly generated by the server if no argument is given. This password will be uploaded with every token that is registered into the directory.

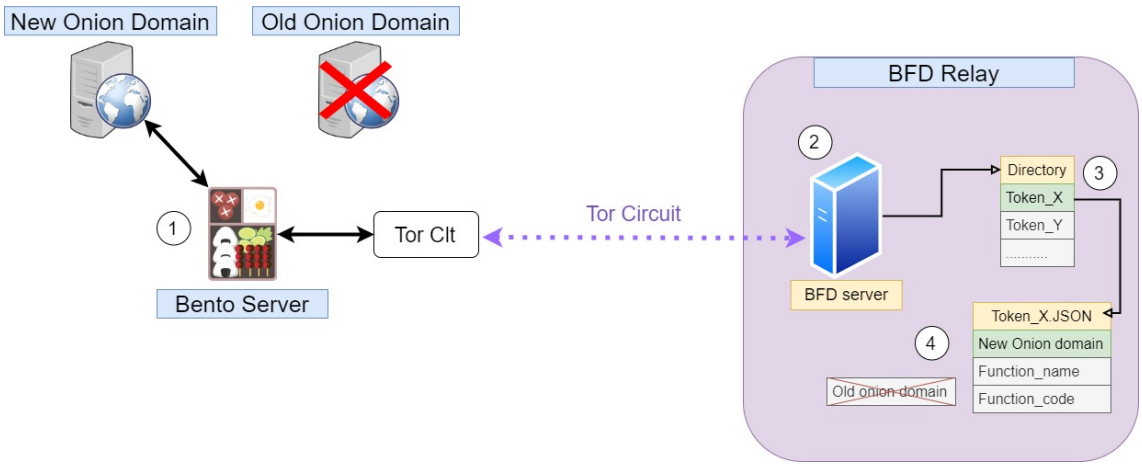


Figure 3.8: A Bento server re-registering an example Token_X with a new onion domain

The BFD server stores this password in another location within the Tor relay, linking it with the given token. If a Bento server changes the onion address of their hidden service, they can re-register the token with the newly formed onion domain by providing the particular token's password to the BFD server.

Figure 3.8 shows us the process of re-registering a token. First (1) The Bento server server creates a new onion url which it then sends to the BFD server along with the token and the password it initially sent with that token. (2) The BFD server than verifies that the password matches the password stored for that token. (3) Then if the token exists and the password matches, it looks up the details of that token and (4) replaces the old onion address with the newly sent one. The server then sends a confirmation that the token was re-registered. The token and its function details will remain the same in the directory, but with it's onion domain now changed to the newly generated domain This way, users can continue accessing a function with the same token if the Bento server wishes to publicly list their onion service again.

3.3.3 Hashing functions and Bcrypt

More on the BFD server side, the password is not stored in plain text. Many studies have shown that this is an insecure way to store password information [10][4]. If the BFD is compromised, an attacker would know the password of all tokens and lead to malicious events, such as redirecting popular tokens to their server, and run a malicious function the client did not intend to run. Thus , when the BFD receives any given password for a token, it saves the token and links it to the **hash** of the password.

The hashing process starts with generating a "salt". The salt is random additional data combined with the password before the hashing process begins. It allows us to transform the data into a longer random string to make it more difficult to unveil the hash and find the true password via brute force methods or dictionary attacks[20]. We then begin to use the cryptographic hashing function known as **bcrypt** [30].

bcrypt was introduced by David Mazières and Niels Provos and uses a one-way hashing function. Which means that the hash cannot be reverted back to it's original form. It

is used here due to its slow encryption process, which makes the encrypted password resilient against attacks more difficult to break compared to hashes such as the SHA-256 hash[9][32]. We use the bcrypt function to protect Bento server tokens in the following way:

1. The Bento server sends the token details and server password to the BFD server.
2. The BFD server generates a `salt` for the `password`, and provides both, along with a numeric `cost` factor to the `bcrypt` function
3. We quickly go into the `bcrypt(password, salt, cost)` function process:
   (a) We immediately start with another function based on the Blowfish cipher[27] named `EksBlowfishSetup`. Which takes in the same inputs, `password`, `salt` and `cost`
   (b) This function acts as the set up phase to ensure a unique hash is outputted from the **key**(`password` + `salt`). The function processes the key through a variant of the Blowfish scheduling algorithm.
   (c) The intensity and number of iterations the key set up phase undergoes, is determined by the `cost` factor initially given. The higher the cost, the slower the hashing process and the more resilient the final output is.
   (d) Once the key set up phase is complete, the actual hashing computation begins. Bcrypt will use the standard Blowfish cipher to encrypt a constant string 64 times with the output of the key set up phase[30]. We call this the `ciphertext`.
   (e) Finally we take the `salt`, `ciphertext` and `cost` and concatenate them to form the final hash.
4. The BFD server then stores the final computed hash in its database, separate from the directory. Associating the hash with the given token.
5. Then any subsequent registration request for a given token, will require its password. Which is verified using bcrypt again, against the stored hash

To practically make use of the bcrypt hashing function, we can use the available Python bcrypt library to implement it within our BFD server. This enables us to enhance the security of the password management system within the Tor relay. The increased computation time within the above explanation of the `bcrypt` algorithm is purposeful. If the database of hashes is compromised by an attacker, for any given hash, they must process any guess through the entire bcrypt algorithm. Forcing the attacker to take a considerable amount of time through several guesses. Hindering both dictionary and brute force attacks.

Bento servers should still securely hide their passwords, as a compromised password will allow malicious attackers who obtain them to re-register their tokens to redirect users to their hidden services. Anonymous 2-factor authentication[33] could also be

added onto this system for Bento servers to safely maintain ownership of their tokens. But this is beyond the scope of this thesis.

3.3.4 BFD server and client interaction

This subsection quickly covers how Bento clients use the BFD server and Hidden Bento servers. The BFD server is accessible by any Tor client. Users connect to it via a 3 hop tor circuit similar to how the Bento server as shown in Figure 3.6. However, they are only allowed to request the existing token related to the data. The intended purpose is to just receive the onion proxy for a listed token. But, as mentioned above, onion services can also list the details of the functions.

For Bento servers who wish to constantly alternate their hidden service domain name, Bento clients will have to regularly search the Bento hidden directory for the updated domain name. This will lead to an even larger increase in time taken to connect to the desired Bento server. Since users have to make a 3 hop circuit to the BFD as well as the regular connections needed by to make a connection to a hidden service as shown in Figure 3.6.

The BFD allows Bento servers to anonymously list their token and function in a public hub. But Bento clients will have to gain knowledge of a given function's token, similar to how regular Tor clients find any onion service, through the clear net. With regular onion websites, it would just be an HTTP get request and retrieving a web page. While this could contain illegal or malicious material, Bento clients will have to be extra careful with Bento hidden functions due to the nature of sending information within their POST request to an anonymous entity. The Bento paper mentions the safe operation of functions within enclaves[23], which are execution environments that are trusted to protect the processes running within. So users could possibly send more sensitive information to trusted anonymous Bento servers that have these active.

Another note is that in the current system, the default paths do not have a re-register option. As this would lead to issues with already public tokens being re-registered to another domain and causing malicious actors to redirect popular tokens in the BFD to connect to their servers as displayed in 3.7. But server's could ensure that token's can be re-registered only under their domain or not, this choice is left to individual servers.

3.4 Token Generation

Tokens allow each function uploaded into a Bento server to have a unique identifier which clients can use to find functions and servers can use to safely store them. In the original Bento design, Bento servers generated tokens only for their machines alone. The Bento hidden directory is expected to handle multiple tokens for multiple functions. Practically, this leads to a problem with how we decide to name and generate our tokens so as not to form duplicates. Each of the many tokens the directory holds must be unique, and Bento servers must be able to easily generate a unique token for their function.

We focus on using an ID generator that will allow us to store a practically unlimited number of universally unique tokens, while also avoiding frequent clashes, as this would increase network overhead for both Bento clients who wish to register their new functions and performance overhead for Bento servers who have to regenerate tokens for the same function to obtain a unique one in the registry. To achieve this, we use the uuid package. Specifically the `uuid` module available in Python.

3.4.1 Diving into the uuid package

`0b88a643-5ab1-4`**`4`**`3c-`**`a`**`5fb-fb115ec2b4eb`

Figure 3.9: Example uuid4 token in hexadecimal notation

Universally unique Identifiers (uuid's) are 128-bit labels used to uniquely identify information in computer systems[16]. They are represented as a series of 32 hexadecimal characters in textual representation.That representation is separated into five groups. Figure 3.9 shows an example generated token using version 4 of uuid.

Msb0	Msb1	Msb2	Msb3	Version	Description
0	0	0	1	1	The time-based version specified in this document.
0	1	0	0	2	DCE Security version, with embedded POSIX UIDs.
0	1	0	1	3	The name-based version specified in this document that uses MD5 hashing.
0	1	1	0	4	The randomly or pseudo-randomly generated version specified in this document.
0	1	1	1	5	The name-based version specified in this document that uses SHA-1 hashing.

Table 3.2: All the versions of uuid and their most significant bits(Msb) which identifies them. Source [16]

To generate a uuid, there is not simply one option, but multiple as which to choose from within the uuid module alone to generate our uuid's for our tokens. In Figure 3.9 the red numeric value represents the version number of the uuid generator used to create the token. The blue hexadecimal value in 3.9 stores the variant of the token which describes the layout of the token. There are 5 of these versions available, their descriptions shown in Table 3.2. We compare them and their usability within our system below:

1. **Version 1 and 2**: When these versions generate a uuid, they use the current timestamp at generation and an identifying property of the device such as the MAC address to create within the token. This weakens the anonymity focus of this project, as users could use these tokens to break a server's anonymity by tracking when a token is uploaded to the directory or matching MAC addresses.

Bento servers will have to take an extra step such as waiting to upload tokens as to not break their anonymity. Therefore we disregard these versions when generating tokens.

2. **Version 3 and 5**: In both of these versions, the generated uuid token is deterministic. They are name based uuid's. When given a particular name they are to generate the same token again. We wish to generate multiple completely unique identifiers that are being constantly generated. Using these versions, the name space themselves have to be unique and so does not work to generate tokens that are unlikely to create duplicates.

3. **Version 4**: This version is the one we choose to generate our tokens. The tokens generated here come from pseudo-random numbers. Those numbers are what largely makes up the token and are generated from a pseudo-random-number generator(PRNG). The PNRG chosen determines the likelihood of generating a duplicate and how often honest servers will generate tokens which clash with one another.

Version 4 of uuid (uuid4) is not only unique but anonymous as the bits generated within the token are drawn from a statistical distribution that does not require systems using it to be linked to create unique tokens. Making it especially good for our anonymous Bento service.

As mentioned, in choosing version 4 we must ensure the PNRG can create universally unique tokens, at least on a practical level such that Bento servers can reliably generate a token and not obtain a clash when registering the token in the directory.

3.4.2 The Pseudo random number generator

6 of the 128 bits in our token are reserved (4 to represent the version and 2 for the variant). Leaving us with 122 bits to randomise and generate a unique token. The distribution selected to generate this randomness is a uniform distribution. More specifically for our implementation, `os.urandom` is used in the uuid4 generation process [5].

`os.urandom` is a cryptographically secure PNRG (CSPNRG). Which means that a computer can generate a random number using `os.urandom` and be unpredictable enough for cryptographic use.

Within the `uuid4` function, the `os.urandom` function is used to generate a 16 byte bytestring, which is then transformed into hexadecimal notation of size 32. The reserved bits to represent the version and variant of the token are then altered before finally generating the unique identifier for the function user to utilise.

The uniform distribution gives each of the 5.3×10^{36} unique v4 uuids that can possibly be generated, equal probability to occur. Since each bit of a token has equal probability of being 1 or 0 and there are 122 bits that are randomised. For any one generated token x to be identical to another generated token y, the probability for that token be a duplicate is:

$$P(x = y) = (\frac{1}{2} \times \frac{1}{2} \times .... \times \frac{1}{2}) = (\frac{1}{2})^{122}$$

The value of the power is 122, as there are 122 bits that are randomly within a generated token. To take from the birthday problem, and show the probability of a duplicate token occurring within n generations, we use the following approximation:

$$p(n) \approx 1 - e^{-\frac{n^2}{2 \cdot 2^{122}}}$$

We will use this approximation to put into perspective the chances for a token being duplicated. The chances of a duplicate token existing within a set of 2^{50} tokens is approximately 1.192×10^{-7}. In other words, if we generated 350,000 tokens a second for the next 100 years our probability of having a token clash is approximately 1.192×10^{-7}.

Practically, this serves our system's purpose. While the tokens are technically not truly unique, they are highly likely to be. And with honest Bento servers who generate tokens, with the `uuid.uudi4()` method, we can reasonably expect no clashes to occur when tokens are uploaded to the directory. Allowing functions for all tokens to be submitted.

An important caveat is that the system works, if the Bento servers are honest and use the`uuid.uuid4()` method to generate their tokens. Section 3.4.3 will approach the issues when this is not the case, and how we handle them.

3.4.3 Alternative approach to handle tokens

In the current system, tokens are generated within Bento servers as they previously would. The Bento servers would then upload these tokens directly to the BFD server. We have shown that the uuid4 package can generate tokens that are unique across systems due to the low likelihood of duplicates. But a challenge discovered here is, since the Bento servers are generating the tokens, the BFD will have to re-verify and ensure that these tokens are uuid generated and not duplicates.

An important question to address is, why not generate these tokens at the BFD instead. To do this leads to an increase in complexity for not only the BFD, but the clients and Bento servers as well. We deal with two approaches for this:

1. Generating One token for BFD and Bento server - The Bento server sends its token to the BFD and the BFD only records its function details, password and onion address. It then generates a new uuid4 token with those details, which it then sends back to the Bento server. The Bento server now has a new token which it replaces the old token with. Then is able to share this token publicly.

2. Generating one token for the BFD and another for the Bento server - This is similar to the above system, except the BFD saves the Bentos server token with the rest of the function details, password and address. Then returns the new token

back to the Bento server. Bento servers will now share this new token so users which clients can look up in the directory.

With both approaches, the BFD is able to ensure it's tokens are uuid4 generated. But for the 1st approach, we see that it removes the previously stored token. If any particular token was used before registration and were later on registered, the old token would no longer exist and would lead to confusion between token holders which Bento servers will have to clarify. Frequent communication such as that may lead to more anonymity issues, it would be ideal to have Bento servers need to make as little communication outside of our system as possible to enhance their anonymity.

The second approach avoids this, as the original token is maintained. Yet, a new issue is introduced due to having two tokens to link to one function. The client has to remember the BFD token to receive the original token and the domain. They can then continue using the original token until the domain is changed again. They then will have to get the BFD token to find the new domain. Bento servers also have to keep track of two tokens for a registered function. This is largely inconvenient to use, and adds additional steps for both the Bento server and Bento client to run one function anonymously.

3.4.4 Verifying tokens

These two approaches alter how Bento servers handle tokens originally. Leading to consistency issues and the need to introduce error handling on both the side of the client and the server. The current system leaves the burden only to the BFD and allows for a Bento server and client to only need to keep track of one token.

By allowing Bento servers to send any string as a token and verifying it before registration prevents incorrect or duplicate tokens from being uploaded, but allows honest Bento servers to generate uuid4 tokens and expect no errors or extra steps when registering them.

The BFD server will ensure that every token it receives first follows the correct form for a uuid4 token, with the version number and variant following the format mentioned in section 3.4.1. It will then have to look through the list of tokens already registered and verify the token submitted is not a duplicate. This adds extra overhead for the registration process compared to the aforementioned approaches, but does not add any new steps for the Bento server or client side to follow. Making this approach the one we implement in our design.

Chapter 4

Testing & Results

This chapter aims to test the performance of Hidden Bento services against original Bento. It also tests the properties of Bento which may break anonymity of Hidden Bento services.

4.1 Analysing speed of Connection

The additional connections introduced in this new design will increase the network latency when sending any kind of request to a Bento server. We first look at the time taken to successfully execute functions already stored in a server, using the hidden service method against using a 3 hop tor circuit and without using tor at all.

```
import requests
import zlib
import os

def browse(url, padding):
    body= requests.get(url, timeout=1).content
    compressed= zlib.compress(body)
    final= compressed
    if padding - len(final) > 0:
        final= final + (os.urandom(padding - len(final)))
    else:
        pad = (os.urandom((len(final) + padding) % padding))
        final= final + pad
    api.send(final)
```

Figure 4.1: Example code snippet a function to send to Bento

The setup for this experiment will be using a single EC2 Ubuntu server to host a Bento server, which stores our target function with a specific token along with 433 other tokens. We then use a personal laptop to send an execution request to this server using

the Bento client API. The requests sent through tor will have the Tor client running , and requests will be sent through the Tor SOCKS5 proxy (via default port 9050) when accessing its network.

For this evaluation, we will mainly focus on executing one function, that is the `browser` function as show in Figure 4.1. The URL fetched is ”example.com/?=ultrasurf” with no padding. We will use this function when referring to execution requests. Testing the time it takes for a client to use it under different connection methods. The goal of this function is to run on a bento server to retrieve a webpage on behalf of the client. Using `api.send()` the function can return data back to the client.

We cycle through 10 iterations for each type of connections to a Bento server. Hidden service connections, regular 3 hop circuit connections and connections without using Tor but directly through the clearnet. The hidden service connection refers to connecting to a hidden service for the first time.

Table 4.1 shows the difference in network latency when sending an execution request to our Bento server using the three different methods. Comparing the three means, we see that the hidden service connection has by far the largest latency. This is expected as we are connecting to several Tor relays (see Figure 3.6 and Section 2.1.2) before making a connection to the hidden service.

	HS Connection	**3 Hop Circuit**	**Clearnet**
Mean	22160.09	1164.44	95.69
Median	15905.20	719.24	93.67
Standard Deviation	13462.45	1188.31	5.18

Table 4.1: Comparison of mean and median latency measurements for different network connections in sending a store request to a Bento server. All values are in milliseconds

Analysing the mean, median and standard deviation of the 10 iterations, we see when connections are sent through the clearnet we get the lowest results. The results are also the most consistent, with the standard deviation being the lowest of the three. This changes significantly when we move up and use a simple tor circuit to send our requests.

The difference is 10 times greater than the clearnet connection, with the standard deviation being 200 times greater. Suggesting that the connections are not only slower, but less consistent. This is even apparent with HS connections being 10 times longer than 3 hop circuits and 100 times longer than clearnet connections. With its standard deviation being even larger. The speeds themselves are in such different tiers that a bar graph of the data on its own would not be easy to observe the difference between the types of connections.

Figure 4.2 allows us to compare the volatility of the connections by taking the log of the time of each process. We see that clearnet connections are very consistent, and the difference between the iterations is barely visible, almost akin to a straight line. The 3 hop circuit connect is more sporadic but apart from iteration 3 which is a clear outlier, this connection is far more consistent than the hidden service connections.

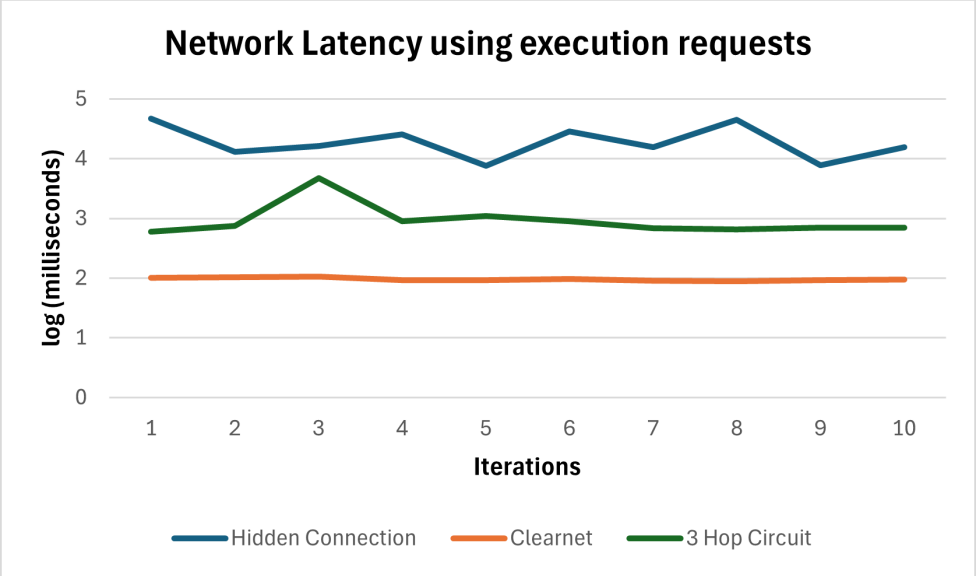


Figure 4.2: Graph which compares the log of the time taken to send a store request to Bento using different connection methods

The hidden service connections are considerably more sporadic, with its values at each iteration being inconsistent. This is probably due to the distributed hash table (DHT) and finding the corresponding onion link within. To initially connect to a hidden service, a user must search for an onion link through the DHT. The time taken here seems inconsistent, with some test iterations hanging for minutes , not making a connection at all. This issue becomes even more apparent when using websocket libraries. Connections would have to be attempted 20 times or more for a connection to be made without timing out.

This is a large usability issue, preventing users from consistently sending requests to servers due to considerably longer wait times to perform a series of requests. However, this issue only occurs when searching for a hidden service for the first time. If a rendevous point is established and another function was to be executed or stored within the same server, the network latency is significantly reduced; Section 4.1.1 discusses this further.

4.1.1 Amortised Costs

The cost of hidden connections shown is expected to decrease with time. The initial connection is the largest part of the cost of connecting to a hidden service. Subsequent requests for the same onion service would be only be sent through a 6 hop circuit. As the rendezvous point is decided, the process does not have to be repeated again.

Table 4.2 adds a new column measuring 10 iterations of sending a store request to a hidden service. But this hidden service at the EC2 server has an established rendevous point with our Bento client throughout the iterations.

We see that the all three measurements under **6 hop circuit** are significantly lower than **HS connection**. The request no longer need to find a way to communicate with the

	HS Connection	3 Hop Circuit	Clearnet	6 hop Circuit
Mean	22160.09	1164.44	95.69	*2779.31*
Median	15905.20	719.24	93.67	*2299.38*
Standard Deviation	13462.45	1188.31	5.18	*1164.99*

Table 4.2: Comparison of mean and median latency measurements for different network connections in sending a store request to a Bento server. All values are in milliseconds

hidden service and so have reduced network latency. The connection is still slower compared to 3 hop circuits, as they are routed through twice as many relays. The 6 hop circuit takes almost twice as long and the standard deviation of 3 and 6 hop circuits are very similar to one another. Showing that connection speed of 6 hop circuits, while worse than 3 hop circuits, is equally volatile. Far less volatile than the latency from forming a connection with a hidden service for the first time.

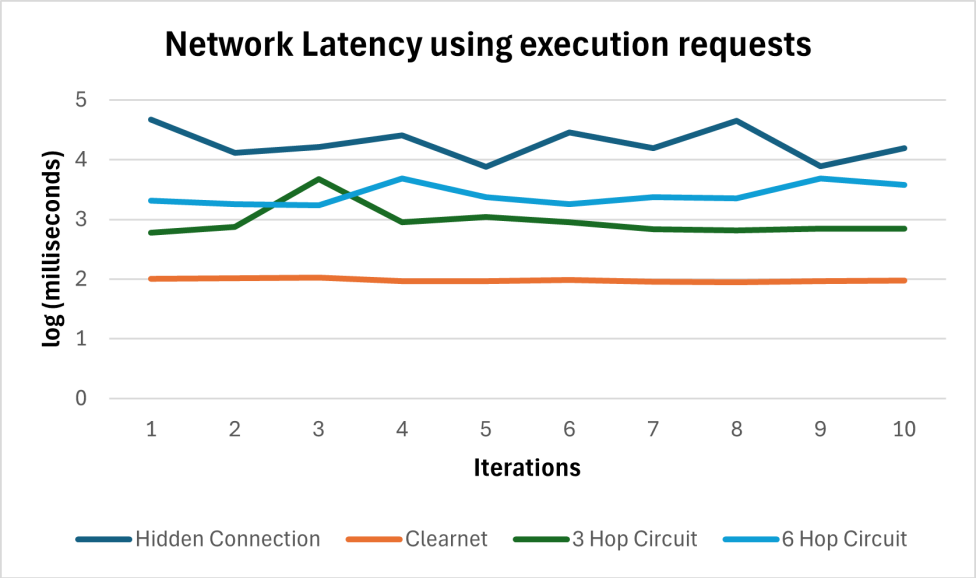
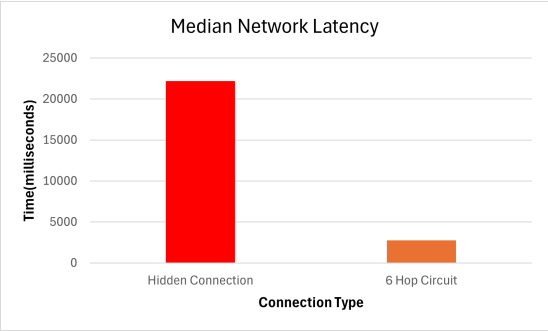


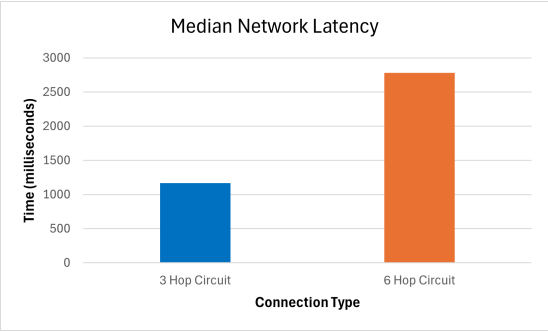
Figure 4.3: Graph which compares the log of the time taken to perform an execution request to Bento with the amortised costs introduced

Figure 4.3 further emphasises this volatility. This graph is identical to 4.2, with the addition of the log of performing an execution request using a 6 hop circuit with an established rendevous point. While noticeably higher than the the time taken for a 3 hop circuit to perform the request, the line is far more stable than establishing a hidden connection for the first time.

Observing Figure 4.4, We also see the latency when using the 6 hop circuit is far closer closer to the 3 hop circuit connection than to hidden connections, showing that the amortized latency of subsequent requests performs far better than it's initial connection suggested.



(a) Hidden connection vs 6 hop



(b) 6 hop vs 3 hop

Figure 4.4: Comparing the median latency of 6 hop circuits, against 3 hop circuits and initial connections to a hidden service

While this section only performs execution requests in its evaluation, these times scale similarly for the other types of requests a client can make. Where the inital Hidden connection will always perform far worse and less consistently than the other connections due to the number of relays it must visit.

4.1.2 Cost from accessing directories

The evaluation could not include the time taken to query the BFD and then connect to the hidden service as shown in figure 3.6. Since we could not obtain a separate EC2 server to host and run the BFD server and hold the directory. The true design is also meant for the directory to be run on a Tor relay, as it is meant to exist only on the Tor network. This is an additional roadblock as there is an additional cost when choosing to run a Tor relay.

Although we did not perform the test of a Bento client making a request to the directory as shown in Figure 3.6 . We can make an estimate using the data obtained in the previous section. Table 4.1 give us the latency cost of sending an execution request over a 3-hop circuit. The Bento client similarly looks up the onion domain of a particular token by connecting to the BFD through a 3-hop circuit.

Excluding the time take to execute the function, the request made by the client to the BFD would be similar to request made by the client to a Bento server through a 3 hop circuit. Both include the time taken for the server to look up the function in their token

directory. The time complexity to do that is $O(n)$ where n is the number of tokens within the server.

The directory for the BFD should be expected to store far more tokens than a regular Bento server due to it storing tokens of multiple Bento servers. If the function related to the execution request is relatively small in processing time, like ”browser”, the time made to complete an execution request would be similar to the time taken to complete an onion domain query for a token if the size of the directories are the same.

Therefore, the latency obtained from a client requesting an execution request for a quick function over a 3-hop tor circuit would be close to the latency obtained from requesting the onion domain of a token, with a similar number tokens within the directories. However, as the number of tokens stored in the BFD increases, the time to query the BFD increases as it searches through a larger list.

4.2 Location revealing functions

While clients can successfully send store requests to Hidden Bento servers by using the appropriate path in the hidden service, this also allows clients to store functions that may compromise the server’s anonymity. This section introduces the possible threats introduced by allowing this feature and how it can be tackled.

4.2.1 Issues with packages and anonymity

The Bento functions can only run on a Bento server if the imported packages have been installed on that Bento server for Python. Some packages give the ability for users to store functions within a Hidden Bento server which can be ran to leak details about the server’s location.

While ”non-anonymous” Bento servers did not have the goal of ensuring functions did not break server anonymity. the original Bento paper[23], they briefly mention the ability of Bento servers to choose which packages they wish to install. Giving them the ability to choose the set of functions they are willing to let clients run on their machines.

This tool is useful to preserve anonymity, but we must be able to examine the packages which would even compromise anonymity and see if it is reasonable to remove packages that allow this. As one of the goals in our design is to ensure that the functionality of Bento is not severely limited.

The next section examines different types of functions that can affect server anonymity, if servers allow store requests and have no extra restrictions.

4.2.2 IP revelaing functions

Figure 4.5 is an example function which uses the socket library to find the IP address of the server. When using a path such as ”/open” shown in Figure 3.4, a Bento client can store this function and then use the path with the function token to receive the I.P.

```
import socket
import zlib

def ip_reveal():
    host_name = socket.gethostname()
    ip_address = socket.gethostbyname(host_name)
    compressed= zlib.compress(ip_address)

    api.send(compressed)
```

Figure 4.5: Example stored function using the socket library to reveal the Bento server's IP address

address of the server. If the socket library is available this function would be performed by the Bento server.

If the module was removed, Bento servers would not be able to use any functions that use the socket library. Yet other simple functions such as the one shown in Figure 4.1 also can be used to reveal the location of a Hidden Bento server. By changing the url parameter for a given call on execution, users can look up pages such as those shown in 4.6. Ultimately, having the requests or any web page fetching library be a risk for anonymous Bento clients to have.

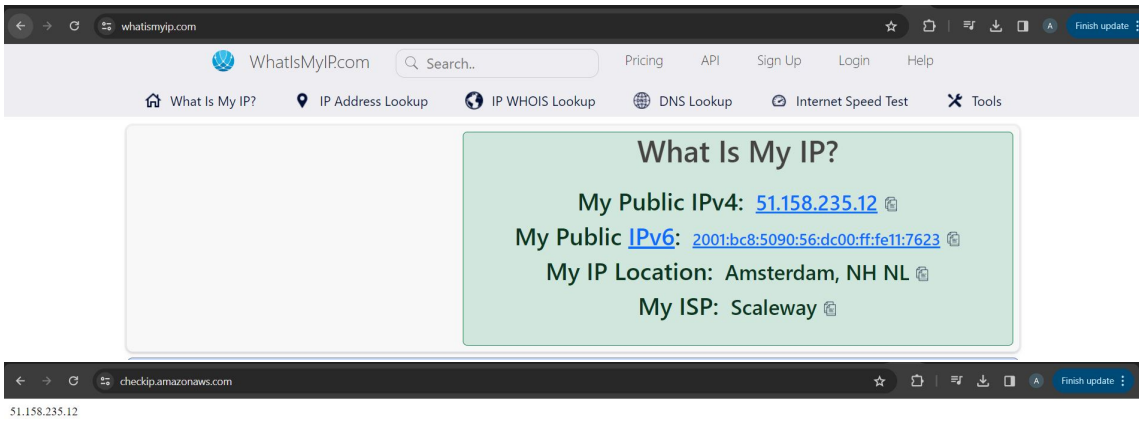


Figure 4.6: Example web pages which can be retrieved by browser to reveal the location of the Bento server

There are multiple webpages which do this and attackers could make their own websites to do this as well, making blocking access to each of those webpages unfeasible. The use of direct HTTP connections in general may also leave openings for other attacks that may leave the Bento server exposed.

4.2.3 Alternative exposing methods

Assuming a Bento server is able to prevent their IP address from being revealed, there are still alternative approaches an attacker can take if a server allows the ability for Bento clients to store their own functions.

Take the `datetime` library in python. This is part of the standard library of Python and has the ability to return the exact local time time of the Bento server using `datetime.datetime.now()`. This value can be used by an attacker,combined with the latency measurements shown in Section 4.1 to estimate the local time of the server and narrowing down the server's location.

Bento clients also dont need to use functions that directly reveal sensitive information on the Bento server's side. Another avenue is by using "network timing attacks".

Suppose a function sent details across the internet to another server, server X. The attacker only needs access to the time at which the server X receives a message from the Bento server, and can calculate the network distance based on the time at which it receives packets from the Bento server. This method also narrows down the location of the server, such as their geographical location and open the gates for other probing attacks.

There may be other possible attacks that can be done due to the freedom that Bento's store ability provides. Disregarding the ability to store functions, trusted developers have to ensure their functions cannot be used to break the system's anonymity. Chapter 4, evaluation, discusses the avenues we can takes to minimise these threats to server anonymity.

Chapter 5

Evalution

This chapter aims to analyse and address the results from the previous chapter. As well as propose solutions to the exploits within the Hidden Bento system.

5.1 Anonymity trilemma

Connections over Tor's regular circuit have relatively low latency , as shown in Section 4.1, compared to other anonymous communication systems, as well as low bandwidth overhead due to Tor's focus to balance anonymity and usability. Tor attempts to optimise performance, but in exchange, tor connections are susceptible to traffic correlation attacks. This is one example of the existing problems when attempting to develop an anonymous system, also known as the anonymity trilemma [34].

The anonymity trilemma states that when creating an anonymous system, it can can largely satisfy only two quantities between strong anonymity, low latency and low bandwidth overhead. Where one of the three is hindered in order to develop an anonymous communication system. This is relevant to our system as well, where we attempt to introduce stronger anonymity, but this must come with a trade off.

Observing the network latency results shown in Section 4.1, the time taken to connect to a Hidden Bento server initially is significantly longer than the time taken to connect to Bento servers with a regular tor circuit. The amortised latency of connecting to the same hidden service is much better and much more stable. But for hidden services that frequently switch their onion domain, the initial hidden service connection may be repeated upon several calls to a given function token.

We are able to obtain stronger levels of anonymity , with bringing anonymity to the Bento servers, as well as with the client due to the extra relays being introduced before connecting to the Bento server. This comes with trade off of higher latency and slightly higher bandwidth due to the introduction of extra relays to establish better standards for anonymity. Following the anonymity trilemma, we cannot expect to have stronger anonymity within our system using Tor, without an increase in either bandwidth or latency.

However, it may be possible to minimise how much they can affect Bento Clients by specific or creative uses of hidden service paths. Looking at the example path in 3.4, the hidden service performs both an execution request and an open request on behalf of the client. This precludes the need for clients to make two separate requests over a 6 hop circuit (i.e., one for an execution request and another for an open request). Bento servers could create paths for frequently used functions, which allow Bento clients to need the least amount of requests to be sent to utilise a function or set of functions.

5.2 Anonymity-compromising functions

Section 4.2 shows us that the fundamental features of Bento can be used to compromise anonymity, since the original design was not built server anonymity in mind. Functions like the ”browser” function are easily able to break anonymity. Bento clients are able to use connections over the internet to attempt to leak the IP of servers or even attempt to leak geographical information using timing attacks. The modules for these attacks have legitimate uses and not having them available within the server, limits the server's functionality and makes Hidden Bento servers far less appealing.

To still allow Bento clients to store and execute requests such as these, Bento servers should not directly access web pages. But should attempt to forward requests through another server to mask their IP address before making connections over the internet.

5.2.1 Utilising Proxies

A proxy server provides a user a way to mask their IP address by acting as an intermediary for a client requesting a resource from another server. Clients send their requests to the proxy which then forwards them to the intended server. For example, Tor nodes in the Tor network are most often used as proxies. By sending requests through the proxy , the functions explored through testing in Chapter 4 would not return the IP of the Hidden Bento server but that of the proxy.

To ensure all the internet connections and requests made by functions are sent through a proxy, the Bento machine must set the system wide connections to be sent over a proxy server. This is possible by forcing any TCP connection made by a function to follow through a proxy. More specifcally, Bento servers can use a tool known as `proxychains` to achieve this.

`proxychains` [22] is an open source software tools that Bento servers can use around executed functions to make sure that they are sent through a proxy and their IP is not leaked. It allows the administrator to choose one or more proxies to send requests through. It supports the inclusion of different types of proxies (HTTP, SOCKS4 and SOCKS5). It also includes the ability to use the Tor network. Allowing servers to execute functions requesting access to the internet and send them through a Tor circuit, if the tor client is running on the Bento server.

Other tools such as a system wide proxy, can also be used. In Linux, administrators can change the `http_proxy` and `ftp_proxy` environment variable of the machine hosting the Bento server, with a proxy server's IP. Making most applications that utilise internet

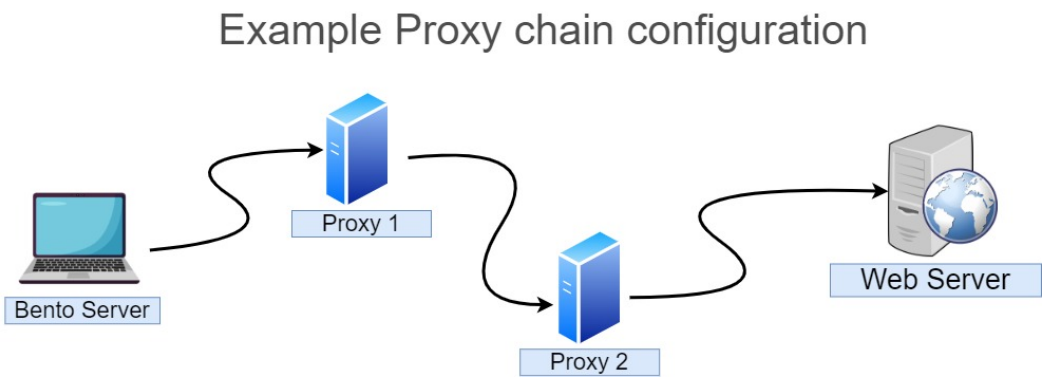


Figure 5.1: A proxcychain configuration with two proxies server being used before accessing an external server.

connections go through the proxy first. The issue here lies with some modules being able to bypass the proxy such as the socket module used in Figure 4.6. In this case, the socket library will have to be removed from the python library, or be used with `proxychains`

Although this reduces the threat of modules allowing for connections over the internet, it introduces additional latency overhead, especially if we decide to use the entire tor circuit as a proxy. We would be adding onto the latency of an already relatively slow system. We must also have to evaluate other modules that wish to be installed, that contains methods which may bypass this system wide setting and leak the location of the hidden service.

An alternative and easier solution is to remove the option for Bento clients to store functions within a Hidden Bento server. Allowing only administrators and trusted clients to upload safe functions to the Hidden Bento server and share their tokens. And allowing Bento clients to use these them via the hidden service's paths.

5.2.2 Removing Store requests

As mentioned in chapter 3, and earlier in this chapter, Bento Servers can create paths to allow Bento clients to access a function over an HTTP connection in different ways. If a Bento Server chooses to only allow Bento clients the ability to execute and open requests, they can prevent clients from storing functions which compromise their anonymity and remove the "/store" path.

Removing the "/store" path will affect our goal of anonymising Bento servers while preserving it's existing features, but allows Bento servers to share their trusted functions to anonymous Bento clients with reduced risk of compromising their privacy. They also can do this without the extra latency that additional proxies bring. This also allows the use of modules which can compromise anonymity. Modules such as the requests module can be used in functions carefully, to utilise their benefits and maintain server anonymity.

Chapter 6

Conclusions

The aim of this thesis was to enhance the anonymity of Bento, maintaining its existing features while introducing a new design. This design allows clients to form connections to Bento servers and be able to access desired functions without disclosing the location of the Bento server themselves. Utilising Tor's Hidden services, we were able to remove the requirement for Bento clients to know the server's IPV4 address. Instead, Bento clients can locate Bento servers indirectly via the hidden service, using its onion domain, which relays information to the actual Bento server.

This advancement allows administrator to host their Bento servers discreetly, being more resilient to de-anonymisation and DoS attacks which aim to censor it's services. Additionally, the introduction of the BFD provides a secure and convenient way for Bento servers to distribute their function tokens for Bento clients to continue having access to. While we were able to provide Bento clients a way to communicate with Bento servers, the reliance of HTTP requests, due to their short-lived connection, is shown to be restrictive. Websockets would be the ideal method of communication but its difficulty to implement in an anonymous way is one of the larger limitations of this system. Even so, due to the freedom given to Bento clients with the ability to Store functions, there is a large set of functions that can be stored to compromise anonymity.

While we can use proxies on the Bento server side to prevent many of these attacks, there still exists modules with methods which server administrators should outright restrict Bento clients from having access to. Hidden Bento servers still provide an anonymous way to share functions that do not break anonymity. And with the additional use of paths, give more control to Hidden Bento servers to what Bento clients have access to without altering the Bento server itself.

6.1 Future Work

Further progress can be made into ensuring Bento servers remain anonymous while providing it's original functionality. As mentioned in chapter 3, work can be placed into attempting to connect to the Bento hidden services using websockets while preserving anonymity. Allowing for clients to have a similar interface as with original Bento.

Additionally, this could also be incorporated with paths and still allow for HTTP communications for more restricted access to servers.

In regards to performance, work is already being done to improve the performance of Tor [7]. Without modifying the underlying system, it is still possible to reduce latency overhead within the network. And possibly even consider optomising the selection of relays to be physically closer, to make the system more efficient [35]. This does bring the issue of affecting anonymity of both client and server but this could be tested and balanced.

This paper was also limited to using one EC2 server and laptop to test the system. Unable to mimic a Bento function directory to test the full system shown in Figure 3.6. With access to more resources, a full replication of the system with multiple Bento servers and clients can be tested and possibly test the introduction of a distributed function directory set up on multiple Tor relays, to better manage load and accessibly of the directory.

This paper also did not include the use of SGX enclaves, which is discussed within the original Bento paper. Since the new design only adds on top of the original, SGX enclaves integrated and tested within hidden Bento servers as well. SGX enclaves may also help solve the issues with allowing anonymous store requests, limit modules within specific enclaves depending on how trusted the client may be.

Bibliography

[1] Stem controller library. https://stem.torproject.org. [n. d.].

[2] Tor project, 2022. URL https://www.torproject.org/. Accessed February 27, 2024.

[3] Alessandro Acquisti, Curtis Taylor, and Liad Wagman. The economics of privacy. *Journal of economic Literature*, 54(2):442–492, 2016.

[4] Erick Bauman, Yafeng Lu, and Zhiqiang Lin. Half a century of practice: Who is still storing plaintext passwords? In *International conference on information security practice and experience*, pages 253–267. Springer, 2015.

[5] Python Core Developers. Uuid module in cpython. https://github.com/python/cpython/blob/main/Lib/uuid.pyL77, 2024. Accessed: 2024-03-31.

[6] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, pages 365–390. 2022.

[7] Roger Dingledine and Steven J Murdoch. Performance improvements on tor or, why tor is slow and what we're going to do about it. *Online: http://www. torproject. org/press/presskit/2009-03-11-performance. pdf*, page 68, 2009.

[8] Jussi-Pekka Erkkilä. Websocket security analysis. *Aalto University School of Science*, pages 2–3, 2012.

[9] Patrick Gallagher and Acting Director. Secure hash standard (shs). *FIPS PUB*, 180:183, 1995.

[10] J. Alex Halderman, Brent Waters, and Edward W. Felten. A convenient method for securely managing passwords. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, page 471–479, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930469. doi: 10.1145/1060745.1060815. URL https://doi.org/10.1145/1060745.1060815.

[11] Tobias Höller, Michael Roland, and René Mayrhofer. On the state of v3 onion services. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Free and Open Communications on the Internet*, pages 50–56, 2021.

[12] Diana L Huete Trujillo and Antonio Ruiz-Martínez. Tor hidden services: A systematic literature review. *Journal of Cybersecurity and Privacy*, 1(3):496–518, 2021.

[13] IETF. *Hypertext Transfer Protocol – HTTP/1.1*. IETF, 1999. URL https://www.ietf.org/rfc/rfc2616.txt. Accessed: March 12, 2024.

[14] Internet Engineering Task Force. Internet Protocol. RFC 791, RFC Editor, 1981.

[15] Felix Lau, Stuart H Rubin, Michael H Smith, and Ljiljana Trajkovic. Distributed denial of service attacks. In *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*, volume 3, pages 2275–2280. IEEE, 2000.

[16] Paul Leach, Michael Mealling, and Rich Salz. A universally unique identifier (uuid) urn namespace. Technical report, 2005.

[17] Viktor Mayer-Schönberger. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*, volume 2. Houghton Mifflin Harcourt google schola, 2013.

[18] Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. Websocket adoption and the landscape of the real-time web. In *Proceedings of the Web Conference 2021*, WWW '21, page 1192–1203, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383127. doi: 10.1145/3442381.3450063. URL https://doi.org/10.1145/3442381.3450063.

[19] OnionBulb. wikimediacommons: InternetSocketBasicDiagram, October 2010.

[20] Benny Pinkas and Tomas Sander. Securing passwords against dictionary attacks. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 161–170, 2002.

[21] Tor Project. rend-spec: Tor rendezvous specification - protocol overview. Tor Project Specifications. URL https://spec.torproject.org/rend-spec/protocol-overview.html.

[22] ProxyChains. How to use proxychains. URL https://proxychains.sourceforge.net/howto.html.

[23] Michael Reininger, Arushi Arora, Stephen Herwig, Nicholas Francino, Jayson Hurst, Christina Garman, and Dave Levin. Bento: Safely bringing network function virtualization to tor. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 821–835, 2021.

[24] R. Fielding Ed Reschke, J. Hypertext transfer protocol (http/1.1): Semantics and content - 4.3.1 get. RFC Editor, June 2014. URL https://datatracker.ietf.org/doc/html/rfc7231section-4.3.1.

[25] R. Fielding Ed Reschke, J. Hypertext transfer protocol (http/1.1): Semantics and content - 4.3.3 post. RFC Editor, June 2014. URL https://datatracker.ietf.org/doc/html/rfc7231section-4.3.3.

[26] Armin Ronacher. Flask documentation. https://flask.palletsprojects.com/en/3.0.x/, 2021. Version 2.0.x.

[27] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption: Cambridge Security Workshop Cambridge, UK, December 9–11, 1993 Proceedings*, pages 191–204. Springer, 1993.

[28] Rachee Singh, Rishab Nithyanand, Sadia Afroz, Paul Pearce, Michael Carl Tschantz, Phillipa Gill, and Vern Paxson. Characterizing the nature and dynamics of tor exit blocking. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 325–341, Vancouver, BC, August 2017. USENIX Association. ISBN 978-1-931971-40-9.

[29] Daniel J Solove. I've got nothing to hide and other misunderstandings of privacy. *San Diego L. Rev.*, 44:745, 2007.

[30] P Sriramya and RA Karthika. Providing password security by salted password hashing using bcrypt algorithm. *ARPN journal of engineering and applied sciences*, 10(13):5551–5556, 2015.

[31] Paul Syverson, Matthew Finkel, Saba Eskandarian, and Dan Boneh. Attacks on onion discovery and remedies via self-authenticating traditional addresses. In *Proceedings of the 20th Workshop on Workshop on Privacy in the Electronic Society*, WPES '21, page 45–52, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385275. doi: 10.1145/3463676.3485610. URL https://doi.org/10.1145/3463676.3485610.

[32] Ruhma Tahir, Huosheng Hu, Dongbing Gu, Klaus McDonald-Maier, and Gareth Howells. Resilience against brute force and rainbow table attacks using strong icmetrics session key pairs. In *2013 1st International Conference on Communications, Signal Processing, and their Applications (ICCSPA)*, pages 1–6. IEEE, 2013.

[33] Ding Wang, Debiao He, Ping Wang, and Chao-Hsien Chu. Anonymous two-factor authentication in distributed systems: Certain goals are beyond attainment. *IEEE Transactions on Dependable and Secure Computing*, 12(4):428–442, 2014.

[34] Chi Xiao, Liqing Wang, Yuping Ren, Shineng Sun, Erlin Zhang, Chongnan Yan, Qi Liu, Xiaogang Sun, Fenyong Shou, Jingzhu Duan, et al. Indirectly extruded biodegradable zn-0.05 wt% mg alloy with improved strength and ductility: In vitro and in vivo studies. *Journal of materials science & technology*, 34(9):1618–1627, 2018.

[35] Guoqiang Zhang, Jiahao Cao, Mingwei Xu, and Qi Li. Less is more: Mitigating tor traffic correlation with distance-aware path selection. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 538–545. IEEE, 2022.