The Pomegranate Router (TPR)

Guillem Larrosa Jara <4lon3ly0@tutanota.com>,
October 2022

Abstract

TPR is an anonymous, decentralized routing protocol. Its purpose is to route a message from Alice to Bob, without Charlotte being able to read the message or discover both parties at the same time. It does not depend directly on encryption for keeping the data secure in transit, and only uses it to protect the identity of the endpoint and validate the identity of the nodes. It routes the message in chunks and the server must rebuild the message once it arrives. Thus, it is needed to use different techniques for making it resilient, like error correction and hash functions. TPR focuses on fragmented and decentralized networks similar to blockchains, and this plus other features render TPR virtually invulnerable to predecessor attacks. TPR provides many of the benefits of Quick UDP Internet Connection (QUIC) while having the reliability of a traditional TCP/IP connection, with the only inconvenience of the higher latency because of the nodes.

Preamble

On regular communications, our identity gets exposed to the endpoint. One of the many solutions we currently have to provide anonymous internet communications is The Onion Router (TOR)[10]. It works by building a chain of intermediaries (so-called nodes) that go from the client to the server (similarly to TPR). That way, the destination only gets to see the last node, and only the first node gets to see the client. TOR also applies onion encryption, an encryption mechanism that wraps messages in multiple encryption layers.

1. Introduction

The main problem with the mentioned Onion Router is that it uses a single node chain. One of its main downsides is that if it gets compromised by a bad actor, they can capture the data. Onion encryption, a multilayer encryption mechanism that TOR features partially solves this issue.

However, the problem with onion encryption (and encryption in general) is that it won't stop the attackers from getting all the data in the first place, as it is not its purpose. Even if encryption prevents a bad actor from reading the message, encryption can be broken. RSA can be cracked[7]. Certificates and private keys can be compromised and/or leaked[17].
Today RSA is uncrackable (in a reasonable time), but as technology advances, it is just a matter of time for it and its successors to be vulnerable. The attackers of high-profile individuals can store encrypted data until they are able to decrypt it. TPR proposes a different approach, where attackers don't get enough of the data in the first place.

I am a Catalan high school student. I am coding the implementation of this protocol in Rust. The only thing that stops me from finishing it is that my 6-core CPU is unable of running all the virtual machines (VM) I need for thoroughly testing the implementation.
I would need 4^3VM for building 3-nodes-long chains while having unique nodes. If I plan to allocate 2 threads per VM, and have a maximum ratio of $5\,\text{VM} : 1\,\text{thread}$, I would need a 32-threaded CPU (which would yield a ratio of $4\,\text{VM} : 1\,\text{thread}$).

2. The concept

TPR works similarly to how stock investors place their investments. They do not bide everything to the same stock. They build a portfolio with many investments, so if one gets lost, the others still have an opportunity to yield profit. TPR uses a similar approach. It builds many node chains and distributes the message in many parts over them. That way, if one node chain gets compromised and a bad actor captures the data, they will only get a meaningless and encrypted little percentage of the original message.

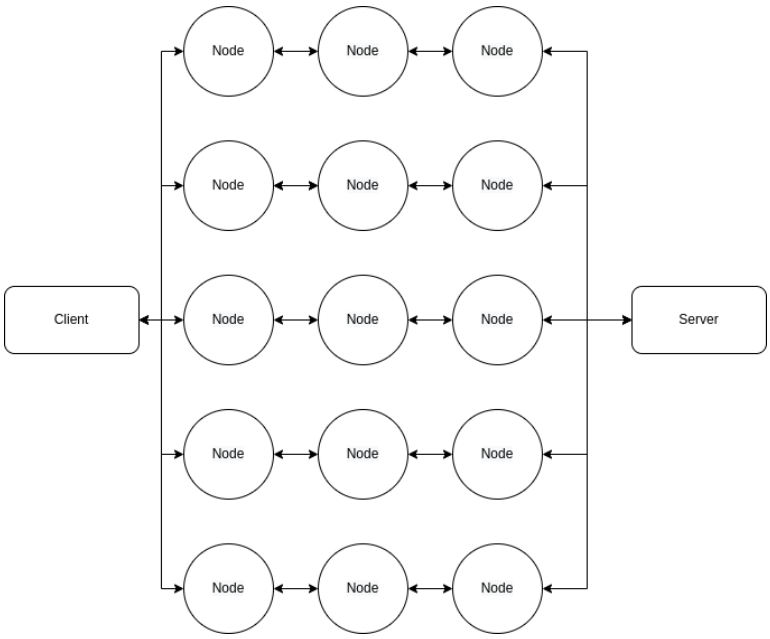

Figure 1: Oversimplification of TPR connection

In contrast to TOR, on TPR networks are more distributed and similar to blockchains[9]. These nodes don't know all the nodes on the network, but only a few. This creates fragmentation and subnetworks, which makes everything harder to trace back to the user from an eavesdropper's perspective. Institutions like privacy foundations or universities might want to host their own TPR node(s).

3. Encryption

TPR uses Transport Layer Security 1.3[15] (TLS/1.3) with *perfect forward secrecy*. Nodes do not use certificates, as they are fundamentally unsuitable for trust-less distributed networks. However, a certificate is the only way to validate the server's authenticity[14], and thus that the guards are not performing a MITM attack, as stated earlier. Connections with servers use TLS 1.3 with preference over certificates, but the client can choose to not use them. If the client knows the nodes they want to use and their public keys beforehand, which is the ideal scenario, they will use forward signatures (section 9.3) to verify the nodes' identity. This can be done for authenticating only the guards, or all the nodes. Thus, and in contrast to TOR, encryption is only crucial for protecting the client from the nodes, but it ceases to be the main way of protecting the data in transit.

Since the connection is split up on several node chains, it is impossible for malicious nodes to neither get significant amounts of encrypted data nor know how much of it they did not get. Encryption is only fundamental for protecting the data at the endpoints, since an eavesdropper would be able to read all the messages. This is important because encryption fundamentally

depends on our lack of capacity of calculating a number *right now*[7], so it is objectively better for it to be one of the layers of protection instead of all the protection.

Once the technology advances enough, encryption algorithms will start to be crackable[7] (and we will need to engineer new ones[11]). Information already encrypted with these older encryption mechanisms and key sizes would now be unprotected[2]. The best way to completely secure a transmission is to stop the bad actors from getting as much of the data as possible, so there is nothing to crack in the first place. With TPR, the only way a bad actor has to get the encrypted message would be to MITM one of the endpoints. Malicious nodes cannot gather useful information, since all the information only useful during the lifetime of the connection is encrypted, and they are on a position where they cannot get any more information (section 13.2).

4. Anonymity

TPR is resistant to predecessor attacks[1]. When a bad actor attempts a predecessor attack, they will have no way of knowing whether they successfully deanonymized the client, as described in section 13.2. However, as section 9.1 states, on network-chosen node chains, malicious nodes can fake being the last ones and, if measures are not taken beforehand, mislead the client into sending the destination. Thus, it is only recommended to enable network-chosen nodes in the case of the guards (and their known nodes) being trusted.

TPR, in contrast to TOR, can afford to let the network choose the nodes. With TOR, a bad actor could capture all the data, but with TPR it would only be a fraction of it. On the other hand, if anonymity is crucial, knowing all the nodes and that they are not controlled by the same bad actor is objectively better. The end user must choose between the added uncertainty on the node chains or more anonymity. Users without a specific threat model will have to compare the trust they have in their guards and how many nodes they know to choose a mode.

5. The network model

Anyone can easily find lists of TOR nodes, as they are public on huge directory servers[10], which are higher-trusted nodes that serve other servers' IP addresses. TPR chooses to use a more blockchain-like approach. On most decentralized blockchains, nodes only know a fraction of the network, a few nodes. When a node receives a new block, it tells all the nodes it knows about said new block, keeping the network up-to-date[9]. If they, in turn, recursively repeat the process, a flood-like propagation effect occurs where all nodes eventually know about the new block.

TPR works in a similar way. Nodes only know a few nodes, and no nodes know a significant portion of the network. While TOR's directory servers are a few who contain the most of the network, TPR prefers nodes to serve as *proxy directories* of the nodes they already know, and leaves the option to the user of choosing their own. This way TPR networks become self-contained, fragmented, and isolated, and thus it is impossible to compromise large portions of the network by compromising a few servers. However, it is a large roadblock for users trying to use TPR, as they must know working nodes first.

6. Synchronization

TPR works in a request-response synchronous HTTP-like process, where Alice sends a message to Bob, and until he answers, Alice may not send more. If Bob timeouts, Alice may perform Automatic Repeat Request[8] (ARQ) up to *n* times, where Alice resends the message that Bob did not acknowledge until he does so. TPR uses TCP sockets, which automatically handle ARQ.

However, the server can switch to an asynchronous connection by either returning status code 102 Processing or status code 202 Accepted. The latter will automatically time out while the former will not.

7. The message

7.1 Fields of a TPR message

Any message sent to the client or the server must contain the following fields:

- Status code, when it is the response status from the server (Appendix A).
- Range: Two unsigned 64 bits integers in the set of $[a..b]$. It is used for resuming downloads and partially recovering data. If the range is null, it is equivalent to $[0..responseLength]$.
- Path: Used for requesting specific content to a server. On a URL it is the part after the domain/IP. When using FPI (section 9) it is used for the server's IP instead.
- 64-bits reserved space for future fields.
- Content: The actual message.

In order to prevent OOM on the server-side (especially during Denial Of Service (DOS) attacks), it is needed to have a maximum path length and maximum content size enforced by the server. This limits the maximum message size to an arbitrary and context-aware:

$$content_{maximum} + path_{maximum} + (64\,\mathrm{B} \cdot 3)$$

7.2 Message states

TPR's name comes from the pomegranate, as it resembles how it treats messages. On TPR messages have two states:

- The Pomegranate itself. This is the complete and readable message.
- A collection of seeds. These are the ones sent across nodes and ought to build a full pomegranate on the server at some point.

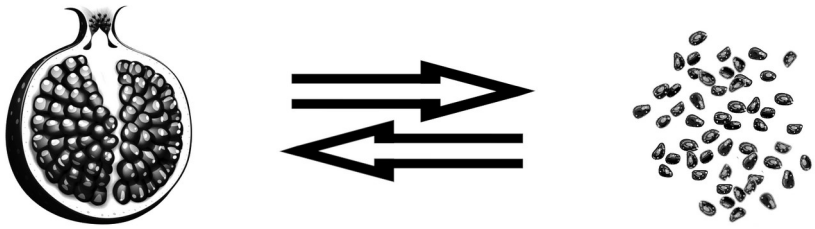

Figure 2: The full message (the pomegranate) and the chunks (the seeds)

7.3 Message conversion

There are 2 ways of converting a message into seeds, which are negotiated during the client-server handshake. The first is the preprocessing:

- Compress, if enabled. This reduces the number of generated seeds.
- Encrypt the message with the server's key.
- Apply error correction if enabled.
- Split the message in chunks *n* bytes long.
- For every chunk, attach the metadata described in the next section.
- Encrypt the part of the chunks nodes don't need to read, if any.

The second one is the QUIC-like[12] postprocessing:

- Apply error correction if enabled.
- Split the message in chunks *n* bytes long.
- For every chunk:
  - Compress the message part, if enabled.
  - Encrypt the message part.
  - Attach the metadata described in the next section.
  - Encrypt the metadata if needed.

The advantage of the postprocessing approach is messages can start to be loaded before having completely arrived. This is useful on applications that can be partially processed, like websites. It is preferred to use the preprocessing approach on communications where the message is big and will take advantage of better compression, while it won't take advantage of load-before-completion. This is the case with file sharing and text messaging.

8. The seeds

This is the second state of the message, the one that is used to send it across networks. Seeds contain the following elements, all of them except two are encrypted with the symmetric key resulting from the TLS handshake with the server:

- An *n* bytes long slice of the message.
- A 64-bits-long index (its position in the message).
- A 16-bits-long message counter, so old seeds are not placed on new messages.
- A 64-bits-long size hint for the server, so it can know the size of the message.
- If the node chain is being built and/or the connection is being resumed:
  - The server's IP address, it is encrypted with the end node's TLS key.
  - A 128-bits-long UUID, it is encrypted with the server's shared public key (section 10).

The UUID's purpose is to allow the server to match the symmetric key from the TLS handshake and the seeds together, so it can to decrypt the seeds and build the message. It also serves session resumption purposes (similarly to TLS' tickets and PSK) and on-the-fly addition and subtraction of node chains.

The message counter serves two purposes. The first one is avoiding slower node chains to place old seeds on new messages. The second one is making it more difficult for an attacker to hijack a connection using the connection resumption feature with a cracked/leaked UUID. If a server detects lots of seeds with different message counters from a specific node chain, it will assume an attacker is brute-forcing the message counter and will close the node chain.

8.1. Avoiding timing analysis attacks

Timing analysis[10] is a kind of attack where an eavesdropper is able to watch both endpoints. By measuring latency, amounts of messages, and message sizes during sustained periods of time, they become able to tell whether both ends are communicating with each other or not. Thus timing attacks deanonymize both parties. TPR partially avoids this by hard-coding the total size of seeds. It ought to be a power of two of any bit magnitude, up to 1024 (e.g., 2 KiB, 512 MiB). Null padding may be added before encrypting to fill the size requirement, as null bytes at the end of the seed will be stripped. Seeds being of regular sizes and the noise from mixed connections severely lowers the chance of successful timing analysis attacks, though they still are a threat.

9. The nodes

TPR nodes act as simple entities. They do not communicate between each other for anything that is not essential for routing the actual data. Distributed networks like the one this paper's proposes can't take advantage of network-level congestion control and other network-wide features that other more unified networks like TOR's can[5].

TPR uses multiple guard nodes. The client gets the best anonymity since these act as an opaque curtain, hiding their identity from less trusted and potentially malicious nodes. Guards also keep the data safe, as none of them get a significant amount of the message. Thus, capturing the data would require all the guards to be controlled by the same bad actor. When it comes to this, on TOR the guard is the single point of failure. On TPR, there are multiple guards, and thus multiple points that need to fail at the same time for the encrypted message as a whole to be captured. It is worth noting that onion encryption also makes guards be the ones who have to decrypt the most layers, but TPR also includes this feature as an opt-in. Thus, an onion-encrypted TPR connection is objectively safer than a regular TOR connection when it comes to capturing the message. The only realistic way for a TPR message to be captured would be to wiretap the client's connection to the internet, which is something out of the control of a network protocol.

9.1 Network-chosen node chains vs client-chosen node chains

TPR offers two main ways of routing the information; client-chosen node chains and network-chosen node chains. On client-chosen node chains, the client selects which node, at every depth, routes every seed. The authenticity of the nodes and the secrecy of the connection are validated by forward signatures (section 9.3). On the other hand, on network-chosen node chains, the previous node selects the next one by randomly choosing a known, and presumably trusted node. If the TPR network is structured as this paper proposes (TPR can be compatible with a TOR-like network), nodes will often know more nodes than the client. Because of this, if the guards are trusted enough, a client may decide to let the guard and its subsequent nodes build the chain.

If the client opts to use network-chosen chains, nodes will build the chains via a non-deterministic approach (section 9.2). However, network-chosen chains are vulnerable to bad actors misleading the client into sending the destination, as they just have to tell they are the last node. That is the only trust given to network-chosen chains. If the endpoint's anonymity is crucial for the client, network-chosen node chains ought to not be used.

Client-chosen node chains are more secure, as node identities are validated and encryption is more robust. However, they are more predictable as they do not have the advantage of non-determinism, and clients need to know significantly more nodes. They also do not take the advantage of nodes serving as directory servers.

As the Figure 3 shows, it is possible to combine client-chosen node chains and network-chosen ones, and get the best from both options. Clients may want to choose their last node so that forward signatures protect the destination, and network-chosen nodes cannot mislead the client into sending the destination, as they would be unable to read it. Clients would send a request to the (allegedly) last network-chosen node to route to a client-chosen node, and said node would use forward signatures before sending the request to route to the server. Because at every depth, the client chooses whether the next node is network-chosen or client-chosen, no information is given to the node, which thus does not know if the chain is completely client-chosen, completely network-chosen, or hybrid.

9.2 Building node chains

When building a network-chosen node chain, a counter is not used for ensuring 3 nodes. This is because of the reasons listed in section 13. Instead, nodes pick a random number between 0 and 3 (there are 4 possible outcomes). If it is 0, the node sends its public key to the client with a request to send the seeds. Elsewhere, the node routes the seed to another node, which does the same.

When building client-chosen node chains, the client performs a TLS handshake with every node before sending the next one or the destination. In order to prevent nodes from impersonating other nodes, the client uses forward signatures (section 9.3) on the TLS handshakes.
Thus, client-chosen chains have the advantage of ensuring integrity on the nodes, and combined with section 13.2 TPR becomes virtually impenetrable.

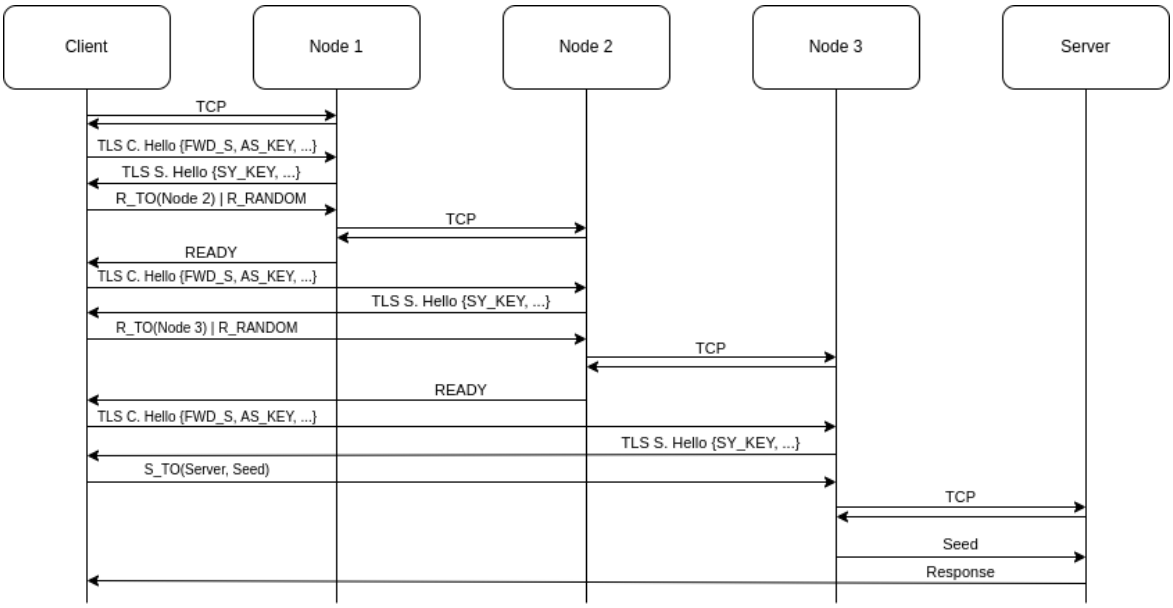

Figure 3: TPR connection. Client-chosen chains use R_TO, network-chosen ones use R_RANDOM.

9.3 Forward signatures

Forward signatures are an essential part of TPR, as they are the most efficient way of ensuring both identity and secure encryption without revealing any information. They keep the TLS handshake to two hellos and do not share any non-ephemeral information, so *perfect forward secrecy* is maintained. A bad actor MITM-ing a client-node connection or a node-node connection with forward signatures enabled will be unable of reading nor modifying any data.

The client speculatively adds a random 256 bits payload (the forward signature) encrypted with the node's known public key to the TLS Client Hello. Once the node receives the Client Hello, it will proceed to encrypt the TLS' generated symmetric key with the (decrypted) forward signature using AES-256, and then it will encrypt the result with the client's public key (like on regular TLS). Finally, the client will receive the node's Server Hello, decrypt the symmetric key with their private key, and then decrypt the symmetric key with the forward signature. It is important to note forward signatures do not verify the node's authenticity *per se*. However, any bad actor impersonating or MITM-ing the node will be unable of reading any of the seeds, as neither party will encrypt with the same key.

As a side note, I am not sure at all if I have reinvented some wheel somewhere in the cryptography field, but it can be described as a non-verifiable zero-knowledge proof with side effects.

9.3 The lack of need of exit nodes

On TOR, exit nodes are special nodes available for being the end node on the chain. These are special for two reasons:

1. They can see all the clearnet servers users connect to. Due to this, they have been targeted by attackers multiple times in the past[6].
2. They can be held legally liable for the data they transport and the servers they connect in some countries[4].

TPR prefers to not make this distinction (except for the next section) due to security concerns. However, since TPR sends the data in chunks, even with backdoored encryption, nodes would have no way to check for illegal content.

9.4 Foreign Protocol Interface

TPR by default cannot be used as a transport layer for HTTP (or any other protocol) without the server supporting TPR in the first place. However, it is solved by using central exit nodes which act as a proxy. They are used as a compatibility agent, so the changes to the protocol are minimal. The only requirement is to encrypt the indexes with the FPI agent's encryption key instead of the actual endpoint's, as it is the FPI agent who must rebuild the message. The message's content must be a valid request of the interfaced protocol since the exit node will forward the request to the endpoint byte by byte.

The FPI agent is *a priori* protocol agnostic. It will expect the path field of the message to contain the IP of the target, the transmission protocol, and the port. Everything else (URL, handshakes, etc.) with the endpoint is handled entirely by the client.

The FPI agent is treated like a regular TPR server. It performs a TPR client-server handshake with the client and receives seeds the same way as a regular TPR server. When receiving seeds, it decrypts the indexes, rebuilds the message, and sends it to the endpoint. Responses will be split up in chunks like on regular TPR messages and will be sent back to the client. Thus, nodes are unable of distinguishing regular servers from FPI agents, and the TPR protocol is kept agnostic.

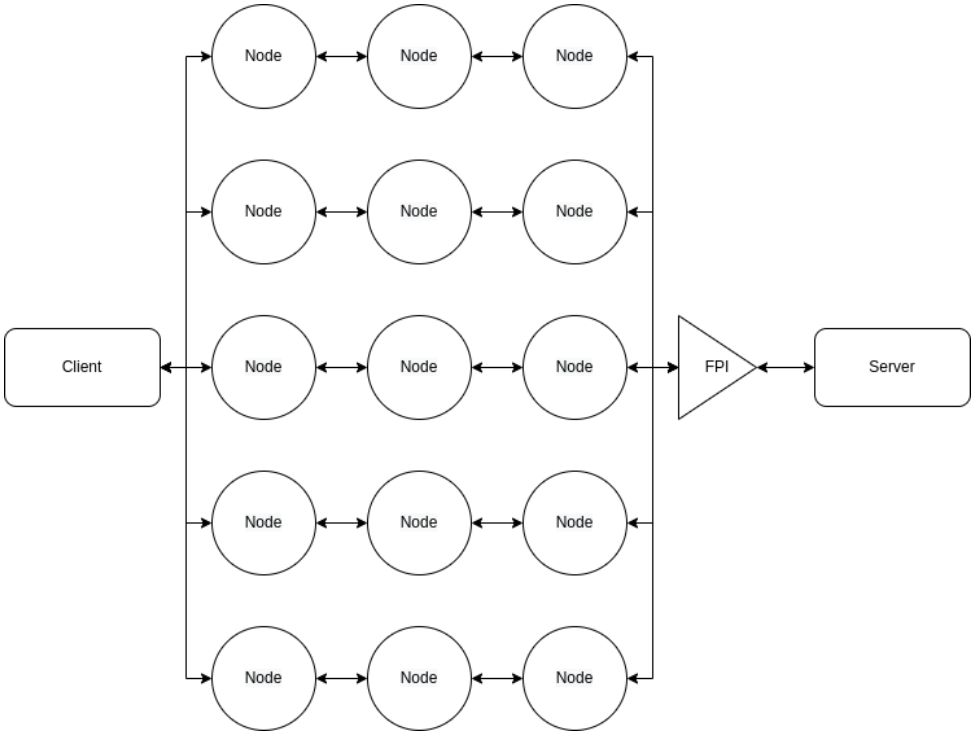

Figure 4: Client sending TPR seeds to a FPI agent, routing to the server.

FPIs act as a single-point failure when it comes to both the message and the endpoint, so users might want to prefer routing protocols like TOR in such scenario. These are made specifically for the purpose of interfacing with protocols like HTTP, and TOR specifically would carry the same advantages as TPR but with less in-transit protection. At the end of the day, it is the users who most compare the available options for their use-cases.

10. The client-server handshake

A TPR connection starts in a single, TOR-like node chain. This is because performing the handshake with the distributed node chains would either require leaking the seed's UUID or, with extra requests, leaking a UUID not tied to the seeds but the client's identity. This would render the client's anonymity broken and thus them identifiable.
Said handshake is a slightly modified version of the TLS 1.3 handshake. It not only negotiates ciphers but TPR-specific settings. The following extensions are added to the initial Client Hello:

- Reed-Solomon Error Correction, which the clients choose between:
  - Disabled.
  - Enabled + amount of redundancy in percentage over the message's size (up to 50%)
- Processing, either of:
  - Client messages will be preprocessed.
  - Client messages will be postprocessed.
- TPR version.
- In the case of messages being postprocessed, a hash function (TLS' will be unsuitable):
  - XXH3: Non-cryptographic fast hash function 64-bits wide with SIMD support[16].
  - XXH128: Non-cryptographic fast hash function 128-bits wide with SIMD support[16].
  - XXH64: Non-cryptographic fast hash function 64-bits wide[16].
  - SHA3-256: Cryptographic hash function 64-bits wide with SIMD support[13].
  - SHA3-512: Cryptographic hash function 512-bits wide with SIMD support[13].

The server then returns the Server Hello with the added extensions:

- A 4096 bits long RSA-OAEP public key shared across all connections.
- A UUID.
- Processing, either of:
  - Server messages will be preprocessed.
  - Server messages will be postprocessed.
- In the case of messages being postprocessed, a hash function (TLS' will be unsuitable):
  - XXH3: Non-cryptographic fast hash function 64-bits wide with SIMD support[16].
  - XXH128: Non-cryptographic fast hash function 128-bits wide with SIMD support[16].
  - XXH64: Non-cryptographic fast hash function 64-bits wide[18].
  - SHA3-256: Cryptographic hash function 64-bits wide with SIMD support[13].
  - SHA3-512: Cryptographic hash function 512-bits wide with SIMD support[13].
- Compression algorithm for both client and server messages, any of:
  - Disabled
  - GZIP
  - Zstandard
  - Brotli

TPR uses an extra public key for the UUIDs since using the certificate would put more trust on the Certificate Authority (CA) than needed, which ought to only be trusted for authenticity verification purposes[18]. It is shared across all the connections to the server and allows encrypting the first seeds for hiding the UUID while maintaining TLS 1.3's *perfect forward secrecy*, as the UUIDs being cracked is not a single point of failure and does not compromise the message's encryption. TLS' symmetric key, TLS' built-in hash function, the encrypted UUID and the message counter protect the TPR connection from hijacking and/or passive eavesdropping.

When postprocessed seeds are used, TLS' hash function becomes unsuitable used for checking message integrity, since it will be checking the seeds rather than the message as a whole. This would thus render said hash unsuitable for checking whether the error correction, if any, recovered the message or whether the message was successfully built. Thus, postprocessed messages need an extra hash function.

10.1 Onion-encrypted routing

TPR is able to provide encryption, where the client encrypts the data with every node's key, and then each node decrypts the data on each hop, removing a layer from the onion[10]. On client-chosen node chains, onion encryption reuses the TLS key from the node TLS handshake, while on network-chosen chains adds the overhead of said TLS handshake.

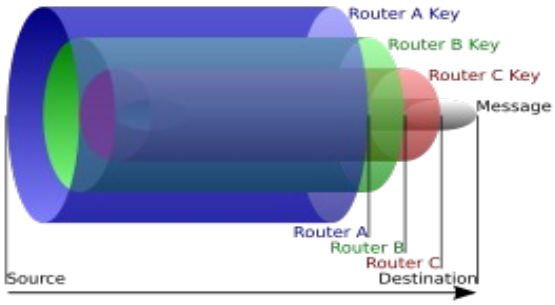

Figure 5: Onion encryption. Source: Wikipedia[10]

End-to-end onion encryption where the end node also encrypts the data with the nodes' keys reveals the amount of nodes in the chain, so cannot be used on TPR. This means that on servers not behind TPR sites (section 12), only client messages can be protected by onion encryption. On client-chosen node chains the overhead of onion encryption is negligible, but on network-chosen chains it severely affects connection build times due to the TLS handshake. Nodes are informed whether onion encryption is enabled via the TLS Client Hello.

11. TPR servers

TPR servers require much more bandwidth, as a single client may create dozens, if not hundreds, of connections. That is the reason why TPR servers must be careful with how many resources they allocate per connection, especially in the first messages. For every new incoming seed, the server must allocate a fixed-size buffer that can store up to the longest URL path and the longest message. Said variables might change depending on if the connection is new or the number of messages sent. If the buffer gets filled and the connection still yields, it will get dropped with status code 413 Payload Too Large.

When receiving a seed from a new connection, the server decrypts the UUID with the shared key and decrypts the content with the associated TLS key. The seed's data field is then saved with its index inside a key-value in-memory database with the UUIDs as keys and different connection settings as values. As seeds carry a size hint, the server will compare it to the amount of received seeds, and depending on the error correction settings, it will decide if it will try to build the message or wait for more seeds. Once the connection has been established, seeds are directly processed if necessary and inserted into its key in the database.

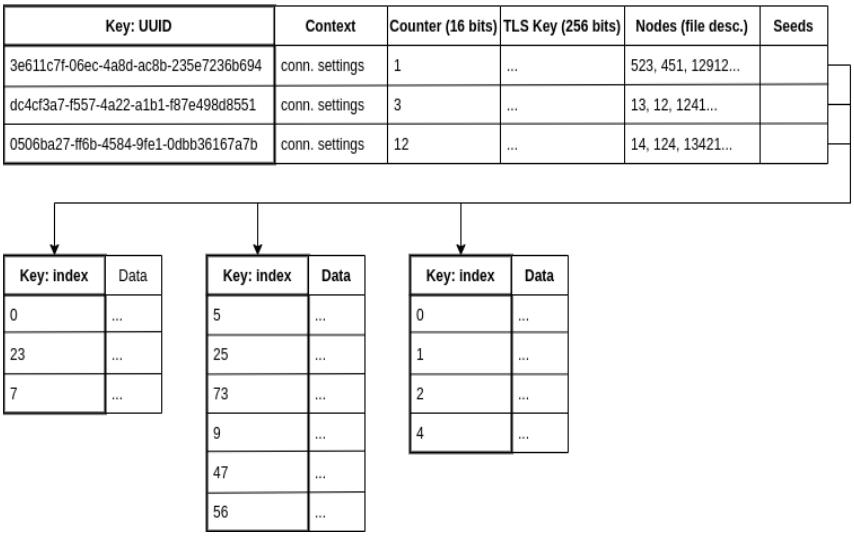

Figure 6: Conceptual model of a TPR server database.

12. TPR Sites

TOR networks have *onion sites*, which are servers hidden behind the network[10]. TPR servers work similarly, by being behind another node chain. This allows implementing end-to-end onion encryption without revealing the number of nodes to the client's chain end node, as there is no need for a node to encrypt or decrypt any data with the other node's keys. TPR servers, at their option, may implement the measures described in section 13 in order to also become indistinguishable from a regular node. Clients must know the public key of the TPR site and an arbitrary amount of nodes (and their public keys) that associate the TPR site's public key to other nodes recursively, until the endpoint is reached.

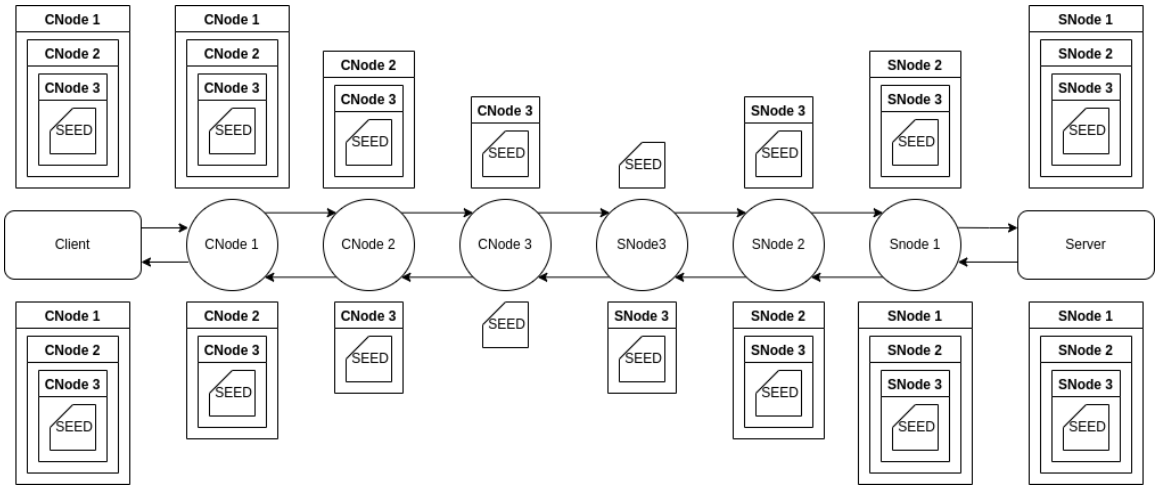

Figure 7: Onion encryption on TPR sites.

As the figure shows, the message is exposed to the ends. While at that depth of the chain both the client and the server are as anonymous as they can be, a TLS handshake ought to be performed before sending any messages in order to not expose the message.

The client builds its chain with R_TO and R_RANDOM frames as usual, but instead of sending a S_TO frame to the end node, they send a R_TO frame pointing to one of the end nodes of the TPR site. Once a TLS connection has been established with said node, the client issues a S_ANON frame containing the TPR site's public key. The node, in turn, repeats the process with the next known node in the chain until the server is reached. The client must know the public key of all the TPR site's known end nodes plus the TPR site's, and every node on the chain must know the key of the former node plus the TPR site's. Each end decides whether they will use onion encryption, as they are responsible for their node chain.

13. Obfuscating clients

TPR by design tries to make as few distinctions as possible between the protocol layers. A great example of this is the FPI (section 9.4), or how both nodes and clients talk to nodes the same way. This section focuses on extra measures for making the client more anonymous.

13.1 TPR over HTTPS

One main downside of TPR is Internet Service Providers (ISPs) could ban its usage or track it down since TPR ought to use a different port to other protocols. A great example of a protocol already implementing this feature is DNS. It can use HTTPS as a transport layer for making it indistinguishable from regular network requests, and thus ISPs cannot MITM the DNS (similarly to DNS over TLS). It is optional for nodes to support carrying seeds over HTTPS on port 80. This would only affect the connection from the clients to the guards, as subsequent nodes may use regular TPR or TPR over HTTPS at their option. The client sends seeds with GET requests, and nodes eventually answer with the server's response seeds.

13.2 Rendering clients indistinguishable from nodes

This is TPR's main defense against predecessor attacks. This is due to to several features of TPR, which combined, makes it impossible for a bad actor to know if they are talking to a client:

1. Non-deterministic node chains: Even if the amount of nodes in a chain is (usually) probabilistic, it being non-deterministic leaves a bad actor without the certainty of having deanonymized the client. They neither know where they are on the chain nor how many nodes there are behind them and (potentially) after them.
2. Same interface for both nodes and clients: Nodes communicate with other nodes the same way clients communicate with nodes, so guards are unable of knowing if they are talking to another node or the actual client based on the behavior of the connection.
3. Distributed and fragmented network model: On TOR, all nodes are public, so a bad actor can easily check if they uncovered a node or not. However, as TPR is fragmented, there is no list with all the existing nodes.

In order to avoid bad actors to test if they are communicating with a client by port-scanning or sending seeds, clients may opt to become nodes during their TPR transmissions. If they ever receive a seeds request, they will know a bad actor is (unsuccessfully) trying to deanonymize them, and it can only be one of the guards.

Conclusion

This paper proposes a routing protocol that keeps anonymity and data protection to the same or higher standards than any routing protocol yet. It does not have any (known) single-point failures, and while it depends on things that can break tomorrow, like encryption and hash functions, if one gets compromised, the other ones still protect enough to consider the protocol safe on most thread models.
An example would be completely breaking all encryption, including client-server TLS. The data would still be relatively protected from eavesdropper nodes since they only receive a fraction of it. An attacker would still need to break hash functions and error correction in order to plug themselves into the connection and MITM it. Encryption prevents both nodes and eavesdroppers at both ends from reading any data. On the other hand, if hash functions get compromised, an attacker would still need to break encryption in order to plug themself into the connection.

A message from the author

Hello. This has been in the making for almost a year now. Even though it has evolved quite a lot, I feel like there are tons of things that can be improved, and I am pretty sure I got some things wrong (the odds are quite high due to the complexity of the topic and my lack of experience because of my age). I'd love to hear your thoughts, so if you want to reach out my email is on the first page. As you probably have noticed, this paper does not go very deep into implementation-specific topics like encoding and serialization, and focuses more on the conceptual perspective. This is because before deciding implementation-specific things, I preferred having it coded first, so I could properly test different options. As I said in the introduction, I can't afford the server I need for continuing developing the code. Because of how this has evolved since I started coding it, I am pretty sure I will rewrite most of the source, which by the way, I learned a lot from writing it. However, I still lack the equipment to test it once it is done, so I would be very grateful if someone contacted me for donating. If that ever happens, I am looking forward to buying the still unreleased equivalent of the AMD Ryzen™ 7950X with 3D cache, or the still unknown AMD Threadripper Zen 4 (if there is not a competitive Intel processor for virtualization). I would also need about 512 GB of DDR5 RAM, or DDR4 if the processor ends up being Intel (their new processors still support it, and it is significantly cheaper).

Acknowledgments

I wish to thank Mike Ashby for their "How to Write a Paper" work, it was a very useful resource. I also wish to thank my internet friends Brendan Falk, Aru and Gonsa for their support.
Special thanks to arXiv for making it so difficult for independent researchers to post papers on your platform that I had to spend 2 weeks emailing random people, just to get little to no response and then give up. Going off-topic, and as a suggestion, you could allow posting papers with the condition of keeping them hidden from the public listing, making them only visible to endorsers or people who have the direct link to it. That way, endorsers can easily endorse good papers without affecting the platform, and researchers can share their research and host it on something other than a Git-based code hosting service.

References

[1] D. R. Figueiredo, P. Nain, and D. Towsley. "On the Analysis of the Predecessor Attack on Anonymity Systems." INRIA. Jul. 29, 2004. https://www-sop.inria.fr/members/Philippe.Nain/PAPERS/ANONYMITY/Pred_Attack_04-65.pdf (accessed: Sep. 18, 2022).

[2] Kaliski and Burt. "TWIRL and RSA Key Size" RSA Laboratories. Apr. 17, 2003. https://web.archive.org/web/20170417095741/https://www.emc.com/emc-plus/rsa-labs/historical/twirl-and-rsa-key-size.htm (accessed Sep. 18, 2022).

[3] Mozilla. "HTTP response status codes." MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status (accessed: Sep. 19, 2022).

[4] M. Masnick. "Austrian Tor Exit Node Operator Found Guilty As An Accomplice Because Someone Used His Node To Commit A Crime". Techdirt. Jul. 2, 2014. https://www.techdirt.com/2014/07/02/tor-nodes-declared-illegal-austria (accessed Sep. 20, 2022).

[5] M. Perry. "Congestion Control Arrives in Tor 0.4.7-stable!". TOR Blog. May 4. 2022. https://blog.torproject.org/congestion-contrl-047 (accessed Sep. 17, 2022).

[6] Nusenu. “Tracking One Year of Malicious Tor Exit Relay Activities (Part II)”. Medium. May 8, 2021. https://nusenu.medium.com/tracking-one-year-of-malicious-tor-exit-relay-activities-part-ii-85c80875c5df (accessed Sep. 22, 2022).

[7] S. Blanda. “Shor’s Algorithm – Breaking RSA Encryption.” AMS Blogs. Apr. 30, 2014. https://blogs.ams.org/mathgradblog/2014/04/30/shors-algorithm-breaking-rsa-encryption/ (accessed: Sep. 21, 2022).

[8] Wikipedia contributors. “Automatic Repeat Request”. Wikipedia. https://en.wikipedia.org/wiki/Automatic_repeat_request (accessed Sep. 17, 2022).

[9] Wikipedia contributors. “Blockchain”. Wikipedia. https://en.wikipedia.org/wiki/Blockchain#Decentralization (accessed Sep 14, 2022).

[10] Wikipedia contributors. “Onion routing.” Wikipedia. https://en.wikipedia.org/wiki/Onion_routing (accessed: Sep. 16, 2022).

[11] Wikipedia contributors. “Post-quantum cryptography.” Wikipedia. https://en.wikipedia.org/wiki/Post-quantum_cryptography.

[12] Wikipedia contributors. “QUIC”. Wikipedia. https://en.wikipedia.org/wiki/QUIC#Characteristics (accessed Sep 26, 2022).

[13] Wikipedia contributors. “SHA-3”. Wikipedia. https://en.wikipedia.org/wiki/SHA-3#Comparison_of_SHA_functions. (accessed Sep. 22, 2022)

[14] Wikipedia contributors. “TLS/SSL server certificate” Wikipedia. https://en.wikipedia.org/wiki/Public_key_certificate#TLS/SSL_server_certificate (accessed Sep 18, 2022).

[15] Wikipedia contributors. “Transport Layer Security” Wikipedia. https://en.wikipedia.org/wiki/Transport_Layer_Security (accessed: Sep 18, 2022).

[16] Y. Collet. “xxHash - Extremely fast non-cryptographic hash algorithm.” GitHub. Nov. 29, 2021. https://github.com/Cyan4973/xxHash/tree/v0.8.1#benchmarks (accessed: Sep. 20, 2022).

[17] ZDNet Community, Z. Whittaker. “PRISM: Here's how the NSA wiretapped the Internet”. ZDNet. Jun. 7, 2013. https://www.zdnet.com/article/prism-heres-how-the-nsa-wiretapped-the-internet/ (accessed Sep. 26, 2022).

Appendix A. Status Codes

TPR uses HTTP/3 status codes[3] with the addition of the range $[600..699]$ for status sent by a node. This range inherits all status codes from the range $[500..599]$ which is the one reserved for servers. This is due to the similar behavior between nodes and servers and how they can fail:

Table 1: Status codes.

Code	Status	Interpretation/Context	Message content
100	*CONTINUE*	Sent after the server expects the client to send further messages.	Message.
102	*PROCESSING*	The server is processing a request that may take more time	Nothing. Eventually

		than the standard timeout. The client will wait until manually closing the connection or receiving a response from the server. If such a request doesn't block the main process, returning this instead of status 100 is functionally equivalent to an asynchronous connection.	another response with the message.
200	*OK*	Success.	Message.
202	*ACCEPTED*	Same as 102, but with automatic timeout.	Same as 102.
303	*SEE OTHER*	The server wants to redirect the user to another server.	Server domain/IP.
307	*TEMPORARY REDIRECT*	The path given can't be served at that moment but can be found in another one in the same server.	Path to content.
308	*PERMANENT REDIRECT*	The path got moved to another location in the same server.	Path to content.
400	*BAD REQUEST*	Invalid request.	Nothing/Explanation.
401	*UNAUTHORIZED*	Sent after a failed handshake	Nothing.
403	*FORBIDDEN*	The user has not enough privileges or has been banned.	Nothing/Explanation.
404	*NOT FOUND*	The path the user provided is unknown to the server.	Nothing.
408	*REQUEST TIMEOUT*	An idle connection timed out without the last message being code 102. Closes the connection.	Nothing.
410	*GONE*	Same as 404, but indicating that the content used to exist.	Nothing/Explanation.
413	*PAYLOAD TOO LARGE*	The seed was larger than the server could accept.	Nothing.
414	*URI TOO LONG*	The path supplied is larger than what the server can process.	Nothing.
416	*RANGE NOT SATISFIABLE*	The server can't send the range of the petition the client asked, most probably that range is out of bounds.	Acceptable range.
426	*UPGRADE REQUIRED*	The TPR protocol version is not the latest.	Latest TPR version.
429	*TOO MANY REQUESTS*	The client exceeded the number of messages it could send to the server in an arbitrary amount of time.	Maximum requests, reset period.
431	*REQUEST HEADER FIELDS TOO LARGE*	Any of the non-message-slice fields of the seed were larger than expected.	Field/s that caused the error.
500	*INTERNAL SERVER ERROR*	The server ran into an error and couldn't send a message.	Nothing/Error log.
600	*INTERNAL NODE ERROR*	The node ran into an error and couldn't send a seed.	Nothing/Error log.