On the feasibility of short-lived dynamic onion services

Tobias Höller, Thomas Raab, Michael Roland, and René Mayrhofer
Johannes Kepler University Linz
Institute of Networks and Security
Email: {tobias.hoeller, thomas.raab, michael.roland, rene.mayrhofer}@ins.jku.at

***Abstract*—Tor onion services utilize the Tor network to enable incoming connections on a device without disclosing its network location. Decentralized systems with extended privacy requirements like metadata-avoiding messengers typically rely on onion services. However, a long-lived onion service address can itself be abused as identifying metadata. Replacing static onion services with dynamic short-lived onion services may be a way to avoid such metadata leakage. This work evaluates the feasibility of short-lived dynamically generated onion services in decentralized systems. We show, based on a detailed timing analysis of the onion service deployment process, that dynamic onion services are already feasible for peer-to-peer communication in certain scenarios.**

I. Introduction

Onion services allow devices to accept incoming TCP connections without disclosing their network location. They are typically used to host services with high privacy requirements like dropboxes for whistleblowers [1], platforms to organize against oppressive governments, or marketplaces for illegal goods. But onion services can also provide a way to enable temporary and anonymous communication between multiple parties, as long as they have a way to exchange onion addresses ahead of time. If new onion services could be created rapidly, it would be possible to use every onion service only for a single message. This would make it very difficult for global passive adversaries to extract any information about communication behavior from a user's network traffic.

To illustrate the argument for dynamic and short-lived onion services, consider the use of instant messaging applications: The majority of them relies on a central server, which is responsible for accepting and forwarding messages between clients. This constitutes a privacy issue because the entire communication pattern of users can be observed at one point (the central server). Decentralizing instant messaging solves that problem, but also introduces a new privacy issue, because direct network connections between communication partners can now be tracked by entities with access to a user's network traffic, including global passive adversaries. This enables simple deductions like finding out when a user sleeps, which can be used to derive the timezone the user is currently located in. But it can also be used to derive social graphs that show which individuals are in contact with each other, which allows identifying a user as a good friend of an already known user.

An idea to avoid this issue was to transfer messages via the Tor network instead of sending them directly. Several projects like Tor Messenger [2] or Ricochet [3] tried to establish instant messaging with metadata-protection but did not manage to capture significant public attention. Other projects like Briar [4] or cwtch [5] which follow the same approach are still actively developed. A renewed effort made by government officials to force organizations to grant law enforcement access to end-to-end encrypted information [6] has strengthened the argument for decentralized communication, because there is no operator of a central server, who could be forced by law to compromise the confidentiality and privacy of all clients. Especially users with increased privacy requirements against public entities like journalists, activists, or whistleblowers have a strong and justified interest in distributed metadata-protecting messaging solutions.

A rarely discussed limitation of such metadata-preserving messengers is the fact that Tor onion services only provide anonymity on a network level, by hiding the IP addresses of both communication partners. Clients still have a unique identifier (usually an onion address), which they must share with all their communication partners. This is an issue because Owen and Savage [7] have demonstrated that the hidden service directory leaks information on when and how often onion services are accessed. In their conclusion, they go so far as to claim that "*it is straight forward to collect and monitor Tor HSs without detection*". For users of metadata-protecting messengers this means that the hidden service directory leaks when and how often they receive messages.

A potential solution on a client level could maintain a separate onion service for every contact and rotate it whenever messages are exchanged. This could be achieved by including an onion address in every sent message to inform the receiving client how it can reach the sender in the future, very much like the double ratcheting protocol updates message encryption and integrity keys with every exchange [8].

To find out if onion services can be used in such a way in the current Tor network, we conducted a detailed analysis on the performance of the onion service deployment process. In this work, we only investigate feasibility from the perspective of clients. If and to which extent the Tor network would be able to support dynamic onion services is the topic of ongoing research.

II. Tor Protocol

Tor is an onion routing technology that anonymizes network traffic by tunneling it through several nodes. A connection established via the Tor network is referred to as a *circuit* and usually consists of three nodes. The *guard node* (also called entry node), a *middle node*, and an *exit node*. Multiple layers of encryption enforce that only the guard node knows the origin of the traffic and only the exit node knows the destination [9].

The overwhelming amount of Tor traffic is used to connect to public websites. Access to Tor onion services makes up only ~1.5% of the total traffic going through the Tor network [10]. Tor keeps track of all nodes currently existing within the Tor network in a consensus. This consensus is created every hour by a group of highly trusted relays, called the directory authorities. Important for onion services is the fact that every 24 hours a new shared random value is generated and published by the directory authorities. Unless otherwise specified, time periods mentioned in this paper refer to the 24 hours a shared random value is valid and when values are derived from time periods, they are derived using the shared random value of the time period.

Onion services were initially created as an example for systems built on top of the Tor network [11] and enabled users to provide services without disclosing their IP addresses or even having a public IP address. It found reasonable acceptance within the Tor community and several projects (for example SecureDrop [1] and Ricochet [3]) were built on top of it. Over time, several critical issues with the previous version (V2) of the onion service protocol were identified, like the use of 1024-bit RSA and SHA1 which are no longer considered secure. Those ultimately lead to the publication of version 3 (V3) of the onion service protocol. Unless otherwise specified all following descriptions refer to V3 onion services.

Onion services enable the operation of globally accessible anonymous services. While the details of the onion service specifications [12], [13] are quite complicated, the basic concept is relatively easy: Servers select introduction nodes, which can be used to contact them. They publish these introduction nodes within the Tor network, so other clients can find them. Clients can then request a connection with the onion service at a rendezvous point of their choice via an introduction point. Establishing the connection via such a rendezvous point provides anonymity to both the client and the server.

A. Creation of Onion Services

Every onion service is identified by a master keypair.

- The private master key grants full control over an onion service. This key is only used to derive blinded signing keys. Onion services use a new blinded signing key for each 24 hour time period. The fact that these keys can be precomputed allows offline storage of the master key, as it is only occasionally needed to precompute another batch of blinded signing keys.
- The public master key is encoded within the address of the onion service. Along with an optional secret, which can be used to protect an onion service with a password, it forms the onion service credential that clients need to know in order to connect to an onion service.

In order to create a new onion service, the host has to first pick three introduction points. Any nodes within the Tor network can be picked for this purpose, by default they are chosen at random. To hide the network location of the onion service, the host establishes connections to all introduction nodes via Tor circuits and keeps them open as long as the service persists.

In the next step, a service descriptor is created to inform clients about the introduction points of a service. Every descriptor is identified by a blinded public key, which is derived from the hidden service credential and the shared random value of the current time period. Most of the descriptor is encrypted, only information needed to store and distribute the descriptor, like version and lifetime is transferred in plaintext. All other fields, like the list of introduction points, are encrypted with another key derived from the hidden service credential and the current time period. If client authentication is enabled, a second layer of encryption is introduced to ensure that only selected clients can read the descriptor.

Once the descriptor has been correctly generated, it must be made available to potential clients. This is done via a distributed hash table (DHT) split across a subset of Tor nodes referred to as the hidden service directory (HSDir). The location of the descriptor within the DHT is determined by hashing the blinded public key, along with the current time period and the replica number. Replicas are used to distribute descriptors randomly across the HSDir. Additionally, a spread is defined to upload a descriptor not only to the single node determined by the location, but also to the closest nodes within the hash table. So, if one node fails, the descriptor remains reachable. By default, onion services use two replicas and spread the descriptor over four nodes, resulting in 8 descriptor uploads for every onion service.

B. Access to Onion Services

To connect to an onion service, a client must first know the onion address, which contains the public part of the master keypair. With that information (and an optional secret) the hidden service credential can be derived. That enables the client to calculate the identifier of the service descriptor it is looking for, by blinding the credential with the current shared random value. Hashing the blinded key along with the time period and replica number tells the client the location of the descriptor within the HSDir. By default, clients randomly contact one of the three HSDir nodes closest to the calculated location.

Any request to the HSDir must be made via Tor to ensure that clients remain anonymous to the operators of the Tor relays hosting the hidden service directory. Once clients receive the desired descriptor, they can decrypt it by deriving the decryption key using the credential and the current time period.

The client then picks a random node within the Tor network as a rendezvous point and establishes a circuit to this node.

Afterwards, it connects to one of the introduction points stated in the descriptor and asks the onion service to meet at the chosen rendezvous point. The introduction point can forward the request through the still open circuit maintained by the host responsible for the onion service. Upon receiving the request, the onion service creates a circuit to the rendezvous point and has its circuit connected directly to the circuit of the client. At this point, a circuit across six relays connects the client to the onion service.

III. Provisioning Time Analysis

To find out if there are scenarios where it is feasible to constantly generate new onion services, we need to quantify the associated negative performance impact. To do this we measure the time between instructing a host to generate an onion service and clients being able to access it. Only if this latency is sufficiently low, it may be acceptable to generate onion services on-demand, which opens up interesting fields of application for onion services.

Our measurement setup is inspired by previous work of Loesing et al. [14] and Lenhard et al. [15], but instead of measuring the time it takes to access an onion service, we measure the time it takes to create one.

A. Measurement Setup

We use the Tor Stem[1] library to generate onion services. Timing information is extracted from the log file created by Tor and event listeners attached via the Stem library. This allows measuring the time of the following events:

- Start connecting to introduction point,
- circuit to introduction point established,
- introduction point ready,
- service descriptor created,
- start upload to HSDir, and
- finish upload to HSDir.

No good solution was found to measure the time it takes Tor to select introduction nodes when creating a new onion service. It seems reasonable to assume that this time is insignificant for the overall latency, but it could be speculated that one host running many onion services could experience deteriorating performance as Tor does not reuse introduction points[2].

All our tests were conducted with version 0.4.3.5 of Tor and ran on a virtual machine running Debian 10, which was monitored to ensure that no local limitations regarding CPU, bandwidth, latency or memory would impact our measurements. The Internet connection (1GBit/s, low latency) was constantly monitored to be working within “normal” parameters, in order to assure that we do not accidentally measure latency effects or outages introduced primarily through our own Internet link.

To ensure that measurements do not influence each other, a new Tor process is completely bootstrapped within a fresh Docker container for every onion service. Our test system runs one test at a time to avoid different onion services impacting each other. To mitigate the effects of possible issues with our Internet connection or the Tor network, tests are conducted in a loop. Every iteration of the loop tests every configuration once. This loop ran more than 1500 times over a period of 10 days to obtain a sufficiently large sample size.

The Docker container specification with our measurement implementation is available at https://github.com/mobilesec/onion-service-time-measurement to enable other researchers to reproduce our measurements.

B. Measured Configurations

As already mentioned, onion services are still in development and can, therefore, currently be deployed in different configurations. To find out if the method of deployment has an impact on the provisioning time of onion services, four different types were measured:

1) V2: A V2 onion service with default parameters: Old, no longer recommended version, which was mainly included to enable comparisons with previous research.
2) V3: A persistent V3 onion service with default parameters.
3) Ephemeral: A V3 onion service with default parameters which can only be created via the control protocol and will only exist as long as the control connection to the Tor instance is maintained.
4) Vanguard: A V3 onion service with the Vanguard [16] extension to harden it against different deanonymization attacks.

C. Results

Fig. 1 provides a good summary of the results of our analysis. The changes implemented by V3 of the onion service protocol have significantly improved deployment times from about half a minute to less than 10 seconds. There are no significant differences between normal and ephemeral onion services, which is no surprise considering that the only difference between those is the persistence of cryptographic keys on disk. The Vanguard extension also shows no significant changes in provisioning time, which is unexpected because modifying Tor’s behavior via the control protocol should actually cause a performance overhead, but is apparently not relevant for our measured scenario.

1) Provisioning Stages: A potentially interesting explanation for the significant differences in provisioning times between V2 and V3 is provided by Fig. 2. It splits the provisioning into three stages:

1) The time it takes the host to establish the introduction points for the onion service,
2) the time it takes to generate a descriptor for uploading after introduction nodes have been established, and
3) the time it takes to actually upload the descriptor.

The first fact to note here is that V2 onion services appear to take much longer to generate their service descriptors. Since there was no obvious reason for such a significant performance difference, we investigated the source code and found that

[1] https://stem.torproject.org/

[2] The official documentation still has an open TODO on picking nodes. However, a review of the Tor implementation revealed that this is the case.

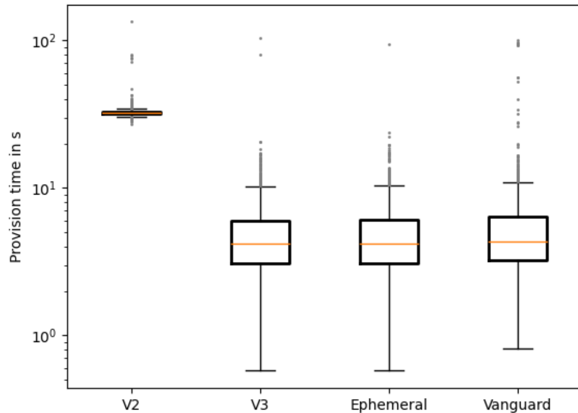

Fig. 1. Overview of provisioning times

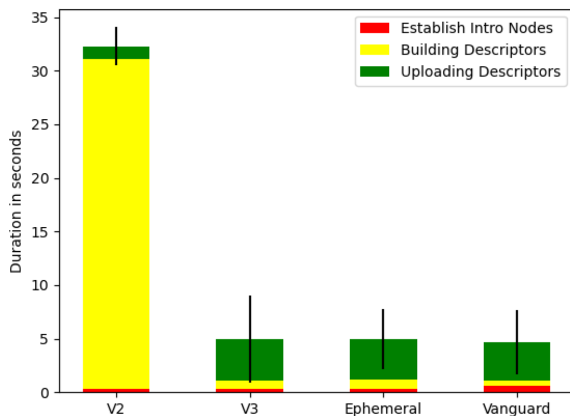

Fig. 2. Composition of provisioning times

the current implementation of Tor V2 onion services waits 30 seconds before uploading a descriptor. There is no explanation in the specification [12] as to why this delay is necessary and the source code only comments that the delay is introduced to ensure that the descriptor is stable. Since V2 onion services are going to be disabled in October 2021 [17], we did not spend additional time on analyzing this issue.

Fig. 2 also reveals other less obvious, but interesting, aspects. For example, it confirms a suspicion that is hard to verify on the logarithmic scale of Fig. 1, namely the fact that for V2 onion services, upload times have not only less impact on the total provisioning time, but are also lower in absolute numbers. The exact reasons for this behavior is analyzed in section III-C2.

Another interesting observation is the fact that establishing introduction nodes is insignificant to the provisioning time of an onion service across all configurations. This observation is however not fully correct because as already mentioned all our measurements were conducted with fully bootstrapped Tor instances. During the bootstrapping process, several circuits (in our experiments we encountered between 2 and 15 circuits during bootstrapping) are prepared, so they can be used for later connections. In our setup, these circuits are always used to connect to introduction points, so our measured time for the creation of introduction points does not include the circuit creation time. Since Tor already collects detailed metrics on circuit creation time [10], there was no reason to analyze them ourselves.

What is worth noting, is that the Vanguard plugin almost doubles the introduction node building time, without impacting the overall provisioning time. At first glance, this seems to imply that Vanguard is actually decreasing the descriptor creation time, which is unlikely considering the fact that Vanguard makes no changes to service descriptors. Instead, the difference is caused by the fact that the generation and derivation of all keys required for creating a service descriptor take a constant amount of time and can already start before the introduction points have been selected. We verified this by deploying onion services with 10 introduction points. Naturally, they needed more time to establish their introduction points, but they still finished creating their descriptors at the same time as services with only three introduction points. This shows that the time needed to establish introduction points is currently irrelevant for the provisioning time of an onion service.

Our final observation is that the descriptor upload is the most significant factor for total provisioning time in current onion service configurations, so we look at them in more detail.

2) Descriptor Upload Times: Our results for descriptor upload times have to be put in context to be understood further: Every onion service uploads its descriptor to several nodes on the hidden service directory. The number can be configured by each service, but the defaults are three nodes for V2 descriptors and four nodes for V3. Both are uploaded in two replicas, so in total there are 6 and 8 uploads. Additionally, the V3 onion service specification requires them to always be valid in two time periods, the previous one and the current one. So when creating a new V3 service from scratch (as done by our test setup) 16 descriptors are uploaded initially.

When Tor clients try to access an onion service, they use their current time period. The previous one is only uploaded to avoid synchronization issues with clients that are still in the previous time period. Tor clients randomly pick one out of only three nodes from one of the two replicas. The fourth upload in V3 is only there to handle situations where a HSDir node goes offline. This means that a single upload could be sufficient to allow an incoming connection. Unless there are any issues with synchronization or failing nodes, six uploads already enable full connectivity. Our measurement setup was not designed to take this into account. Instead, we assume that a descriptor has been successfully published when half of all uploads (3 for V2 and 8 for V3) have been completed. The upload time in Fig. 2 shows how long it took on average to

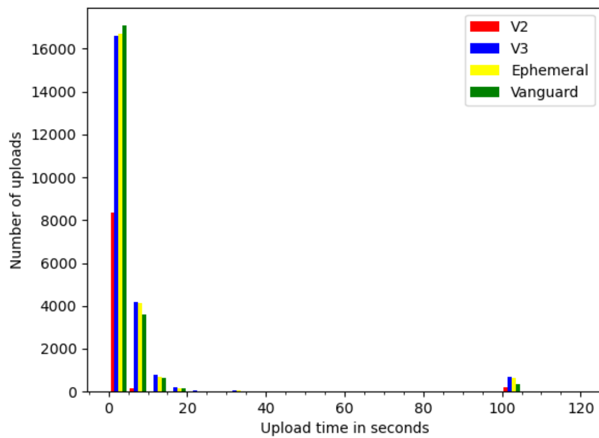

Fig. 3. Time it took individual uploads to complete

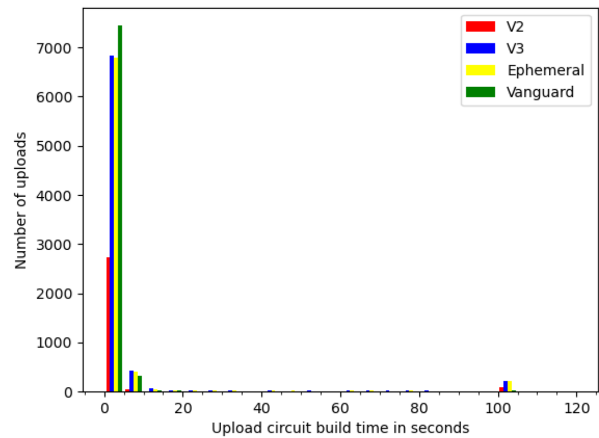

Fig. 5. Time to create upload circuit

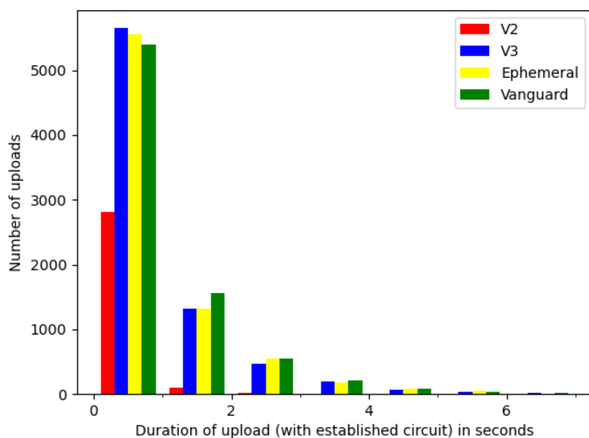

Fig. 4. Time to upload descriptor via established circuit

complete half of all uploads. This decision removes the impact of very slow uploads and tries to find a middle ground between trying to find the earliest time when connections are possible and the time when connections are almost certain to succeed without retries.

Fig. 3 shows the duration of individual descriptor uploads. The majority of upload requests finish in less than 5 seconds and almost all uploads complete after 20 seconds. A noteworthy result of our measurement is an unexpectedly high number of upload requests that take between 100 and 105 seconds, which occurs for all measured configurations, but happens less often for onion services with the Vanguards extension. To further analyze this behavior we conducted a second smaller experiment by running the loop only 500 times and additionally tracking the time when upload circuits were completed. This allows us to split the upload time into the time it took to create a circuit and the time it took to actually upload the descriptor.

Fig. 4 shows that plain uploads hardly ever exceed five seconds and even the slowest single upload we measured only took 12 seconds to complete. The unexplained 100 second delay is only present in the circuit creation time. This makes sense because this delay only happens when Tor fails to open a circuit to a hidden service directory. In this case a 100 second timeout occurs before another attempt is made. This also explains why Vanguards has a positive effect on this issue. It selects a subset of candidate nodes for the second and third hop of a circuit and tries to reduce the risk of picking a malicious node. Apparently, this also reduces the risk of picking nodes, that cause circuit creation attempts to fail, increasing the overall performance and reliability.

Another interesting result in this context is the fact that some circuits fail again after this 100 second timeout. In this case Tor does not wait and try for a third time, but instead abandons the upload attempt entirely. This does not result in any error displayed to the user, because the onion service concept is redundant and a single failed upload has no significant impact on the availability of an onion service. During our experiments we experienced an upload failure rate of about 1% for upload requests without Vanguard and a failure rate of about 0.8% for uploads with Vanguard.

Fig. 4 confirms that V2 descriptors are published faster than V3, which is most likely caused by the much larger descriptor size in V3. Fig. 6 provides a zoom-in on circuit build times below 8 seconds to facilitate comparison with Fig. 4 and shows that the circuit creation time has more impact on how long it takes to publish a descriptor than the actual upload. An interesting observation is that our results seem to indicate that V2 upload circuits are created faster than V3 upload circuits. This effect is again caused by the fact that our Tor binaries were fully bootstrapped before any measurements were conducted, which allows Tor to cannibalize general circuits for uploads if there are any available. Since cannibalization is much faster than creating a circuit from

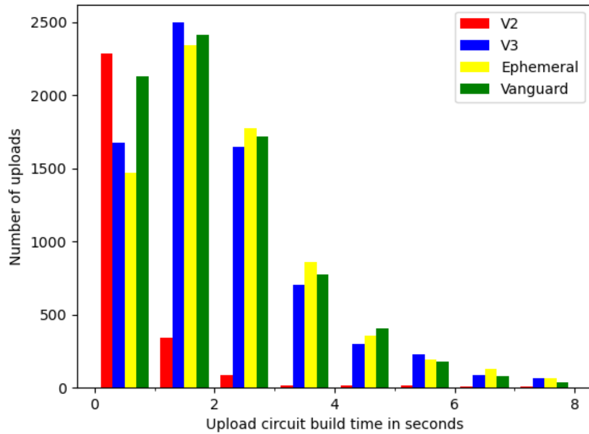

Fig. 6. Zoom-in on uploads circuit build times below 8 seconds

scratch, this means that some upload circuits can be created faster than the rest. The lower number of uploads in the V2 onion service specification increased the relative impact of these cannibalized circuits, creating the impression that V2 upload circuits are created faster. Unfortunately, we could not properly quantify the impact of this issue, so we cannot say if there are any other factors contributing to the increased circuit creation time in V3.

IV. Conclusion

While improved performance was not an explicit objective of the V3 onion service protocol, our measurements show that the deployment performance was improved significantly from more than 30 to about 5 seconds. Despite these improvements, onion service deployment still takes several seconds, which is probably too long for applications like instant messaging where users would experience that additional delay for every message.

We also show that roughly 80% of the time needed for publishing an onion service is spent on uploading descriptors to the HSDir. However, an onion service can also be accessed without being published, if there is some other way to share the service descriptor with the potential communication partner. This reduces the time it takes to prepare an onion service to about one second and reduces the load that the creation of dynamic onion services puts on the Tor network. In an instant messaging scenario, an initial exchange of service descriptors could take place when two individuals are in close proximity by encoding a service descriptor in a QRcode and displaying it for the communication partner to scan. Afterwards, clients could append the next service descriptor to sent messages, thus iterating the used onion service with every exchanged message. This would increase the time it takes to send a message, because a new service descriptor has to be created every time, but an additional delay of a single second might be acceptable for many scenarios. User experience could be further improved by already setting up a new onion service while the user is still typing. Avoiding the hidden service directory has the additional benefit of removing one possible attack vector against communication via onion services.

In conclusion, applications intending to use short-lived dynamic onion services should either be able to tolerate noticeably increased latency or try to find an alternative way of exchanging service descriptors, until Tor further improves the performance of circuit creation.

Acknowledgment

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World, funded by the Christian Doppler Forschungsgesellschaft, 3 Banken IT GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH and has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] A. Swartz, "Securedrop," https://github.com/freedomofpress/securedrop.
[2] sukhbir, "Tor Messenger Beta: chat over Tor, Easily," https://blog.torproject.org/tor-messenger-beta-chat-over-tor-easily.
[3] J. Brooks, "Ricochet," https://ricochet.im/.
[4] M. Rogers, "Briar," https://briarproject.org/.
[5] Open Privacy Research Society, "cwtch," https://cwtch.im/.
[6] USA, Great Britain, Australia, New Zealand, Canada, India, Japan, "International Statement: End-To-End Encryption and Public Safety," https://www.gov.uk/government/publications/international-statement-end-to-end-encryption-and-public-safety.
[7] G. Owen and N. Savage, "Empirical analysis of tor hidden services," *IET Information Security*, vol. 10, no. 3, pp. 113–118, 2016.
[8] M. Marlinspike, "The double ratchet algorithm," https://signal.org/docs/specifications/doubleratchet/.
[9] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, 2004.
[10] The Tor Project, "Tor metrics," https://metrics.torproject.org.
[11] R. Dingledine, "Next Generation Tor Onion Services," DEF CON 25, Las Vegas, 2017.
[12] The Tor Project, "Tor Rendezvous Specification," https://github.com/torproject/torspec/blob/master/rend-spec-v2.txt.
[13] ——, "Tor Rendezvous Specification - Version 3," https://github.com/torproject/torspec/blob/master/rend-spec-v3.txt.
[14] K. Loesing, W. Sandmann, C. Wilms, and G. Wirtz, "Performance measurements and statistics of tor hidden services," in *2008 International Symposium on Applications and the Internet*, 2008, pp. 1–7.
[15] J. Lenhard, K. Loesing, and G. Wirtz, "Performance measurements of tor hidden services in low-bandwidth access networks," in *Applied Cryptography and Network Security*. Springer Berlin Heidelberg, 2009, pp. 324–341.
[16] M. Perry, "The vanguards onion service addon," https://github.com/mikeperry-tor/vanguards.
[17] The Tor Project, "Onion Service version 2 deprecation timeline," https://blog.torproject.org/v2-deprecation-timeline, 2020.