Leopold–Franzens–Universität
Innsbruck

Institute for Computer Science
Research Group Quality Engineering

Human-readable Names for Tor Hidden Services

Bachelor Thesis

(SS2011)

Supervisor: Matthias Gander

Simon Nicolussi

September 20, 2011

Contents

Abstract

Hidden services in the Tor network are an easy way to make a service available to unascertainable users while concealing the operator's identity at the same time. Unfortunately the method for addressing an individual service is quite rudimentary and has never been revised since the functionality became available in 2004. This thesis describes the related problems and researches possible solutions with special attention on their applicability to the Tor network. The resulting insights suggest that a slight modification of Paul Crowley's word lists is currently the best bet for providing a robust and user friendly mechanism to address hidden services. The solution is then implemented with special consideration of secure coding principles and proper quality assurance.

1 Introduction

This thesis is concerned with finding an alternative naming system to the simplistic scheme that is currently in use. In order to understand the pros and cons of the different approaches to the problem, it is necessary to have a quite detailed understanding of the way Tor is operating. The technical diagrams in this section are modified versions of the ones found at the Tor Project's website in the overview section[1] and the hidden service documentation.[2]

1.1 The Onion Router

Tor[3] is the name of both a software and a network that tries to conceal the identity of its users on the Internet. There are numerous contenders that try to achieve the same goal, Tor's differentiating characteristic, however, is that it tries to defend against fairly powerful adversaries like for example ISPs and even governmental institutions. Simple anonymizing proxies (e.g. VPNs) consisting of a single hop are frequently good enough to conceal the location of the user at hand from the visited service, but fail trivially if the ISP of the proxy is malignant.

At the same time Tor is intended to be used as a *low-latency network* nonetheless, where interactive uses like real-time chat or Web are possible without a prohibitive speed penalty. As a matter of fact this premise implies that Tor cannot protect against *global observers*. Such an adversary would be able to watch the traffic flow in and out of the network and can then correlate the streams by timing attacks. There are *high-latency* networks that try to defend against global observers by introducing delays when relaying data, Tor on the other hand tries to defend against global observers by preventing them (e.g. by making the network as heterogeneous as possible).

Tor's key principle is *Onion Routing*. The basic idea is straightforward: there is a large number of *Onion Routers (ORs)* that act as nodes in the Tor network. Their duty is to relay the traffic of the *Onion Proxies (OPs)*, client

[1] `https://www.torproject.org/about/overview.html`
[2] `https://www.torproject.org/docs/hidden-services.html`
[3] `https://www.torproject.org/`

Figure 1: Alice's OP obtains a list from a directory server. The pluses symbolize ORs commonly run by volunteers.

software installed on the computers of the users.

Each OP knows the public keys of each OR in the network (Fig. 1). With these keys the OP can build a *circuit* through the network, i.e. a chain of (currently three) ORs chosen at random (with keys k_1, k_2, ..., k_n). If the user wants to relay traffic over Tor, the OP groups the data from the TCP-stream into cells of 512 bytes. Each cell is then supplemented with routing information (r_1, r_2, ..., r_n) for getting to the next OR and subsequently encrypted with the corresponding public keys:

$$E(k_1, E(k_2, \ldots E(k_n, \text{payload}|r_n)\ldots|r_2)|r_1)$$

This "onion" can then be sent over the circuit. Each OR in the circuit can only decrypt the layer encrypted with its public key and only the last OR will see the payload itself (Fig. 2). The routing information is shared on a strict need-to-know basis: each OR only knows about its direct predecessor and its direct successor. This way the anonymity of the user is protected if at least one OR in the chosen circuit is non-malignant (assuming there is no global observer, of course).

This sketchy description of Tor's operation should suffice for now, the actual mechanisms are far more complicated but not of interest for the remainder of this work.

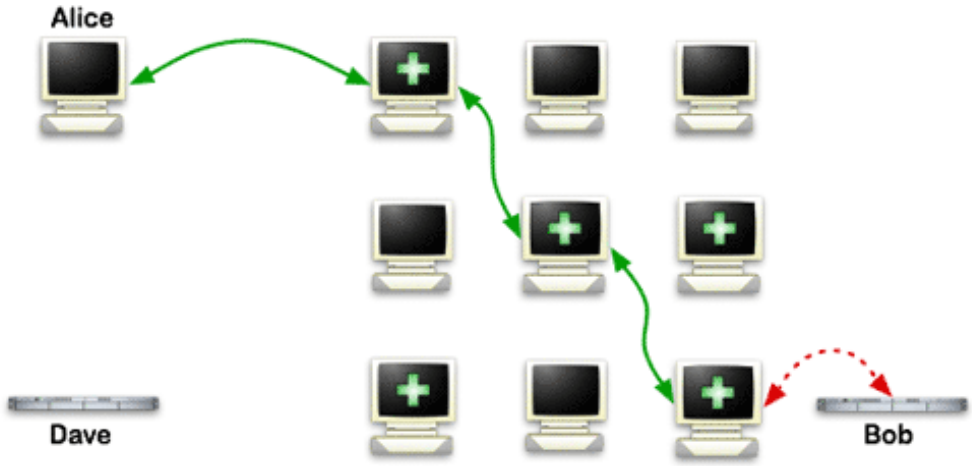

Figure 2: Alice's OP picks a random path to Bob's OP. Green links are encrypted, red links are in the clear. The last link is generally unencrypted.

1.1.1 Hidden Services

Services that are not only accessed by means of the Tor network, but reside inside the Tor network itself, are called *location-hidden services* or, in short, *hidden services*. Residing in the Tor network itself has a number of advantages for the operator of said services, the most important of which is *responder anonymity*. The central point of responder anonymity is that not only the user but also the operator of a service benefits from the protections the Tor network provides.

In ideal circumstances it should be impossible for an adversary to determine:

- where a hidden service is located geographically,
- who operates the hidden service,
- by whom the hidden service is being used,

and, thanks to end-to-end-encryption,

- any communication between user and service is protected.

To set up a hidden service, the operator (say Bob) has to run an OP and an application specific server software (i.e. *nginx*[4] for HTTP). The OP generates a public key and randomly chooses three *introduction points* from the ORs in the network. A structure containing this data (amongst other) is then signed and uploaded to a *distributed hash table (DHT)*. A circuit to each introduction point is built (Fig. 3).

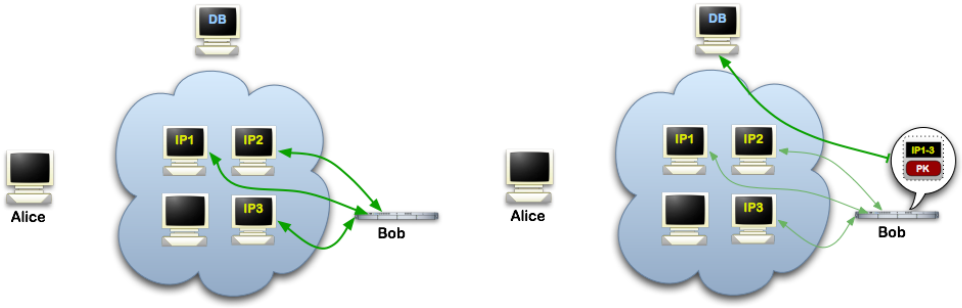

Figure 3: Bob's OP connects to three introduction points and uploads a descriptor to a database (in reality a DHT). Note that the green links are Tor circuits in this context, not direct connections.

A client (say Alice) that wants to visit Bob's hidden service has to download the structure from the DHT. At the same time Alice's OP chooses a *rendezvous point (RP)* by chance from the ORs in the network.

A randomly generated *rendezvous cookie* is combined with the RP and encrypted with the service's public key. Then the OP connects to one of the introduction points and sends the encrypted packet to Bob's OP (Fig. 4).

Now only Bob and Alice know about the rendezvous cookie and the RP. Their OPs build a circuit there and the two connections can be joined under consideration of the rendezvous cookie. Alice and Bob are now connected to each other and their applications can exchange data as via any other TCP connection (Fig. 5).

The biggest disadvantage of hidden services is the penalty in terms of speed. Another problem is the addressing of hidden services. It is of course

[4] `http://nginx.net/`

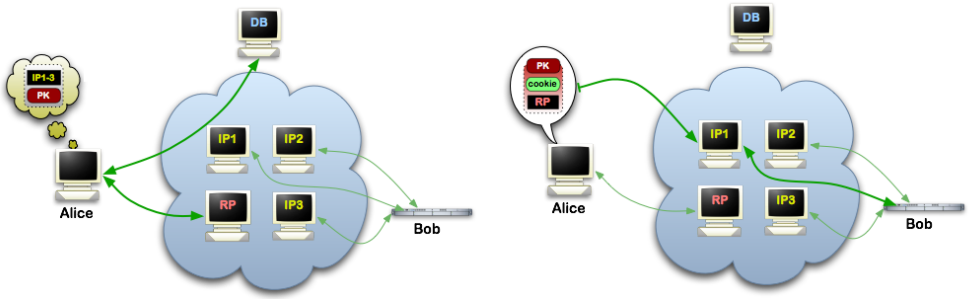

Figure 4: Alice's OP fetches the corresponding descriptor from the DHT and chooses a RP. The rendezvous cookie and RP are sent to Bob's OP over one of its introduction points.

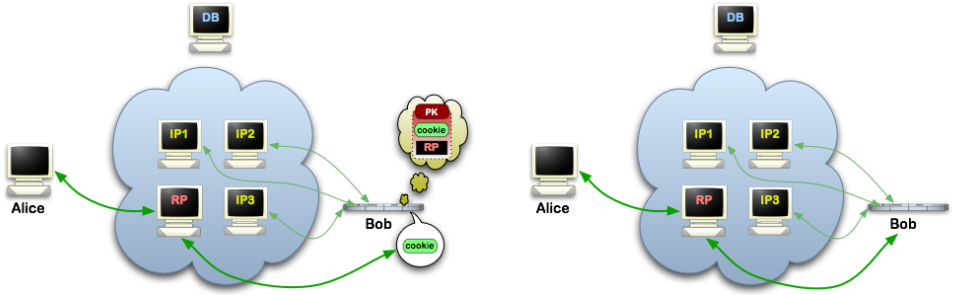

Figure 5: Alice's and Bob's OPs meet at the RP and confirm each others identity with the rendezvous cookie. They proceed to use their circuits like in regular operation.

not possible to address them like a conventional service in standard TCP/IP, as the IP address must remain concealed at all times. The currently used system uses public keys generated by the hidden service in question to establish a connection over the Tor network.

Unfortunately public keys are currently 128 bytes in length (as visible in the Tor source code) and thus unsuitable to be used by humans in any intuitive way. Instead, the public key is transformed to an *Onion URL* consisting of 16 alphanumerical symbols and the TLD look-alike `.onion` is appended to it. An identifier constructed this way can be used just like any URL, provided the user routes his traffic concerning the identifier over an OP. An example of a hidden service URL would be `duskgytldkxiuqc6.onion`.

URLs like these are better than plain public keys in terms of usability,

but they are still a long shot to being as natural as URLs on the regular Internet are. They could also be considered to be a security risk, since an attacker could generate a similar URL and publish it while maintaining the impression that it is associated with the site to be impersonated. Most users wouldn't realize that the URL `duskgyii4e77trop6.onion` is different from the one we presented above, for example. The price of this mimicry was a few minutes of brute-force computations with *Shallot*[5] on a standard laptop for finding an appropriate public key.

The stated goal of this work is to provide a robust system that is more usable and at the same time at least as secure as the currently deployed mechanisms. Even though it looks like a purely cosmetic change, it could have considerable impact on the usage of hidden services among non-technical people, and that in turn makes Tor more secure for all of its users [1].

Furthermore an important aspect is adaptability to various changes in the future; the system should scale reasonably well with the anticipated increases in processing power and contingent structural changes to the Tor network, as for example introduction of elliptic curve cryptography and even longer public keys.

[5] `https://github.com/torzeljko/Shallot/`

2 Problem Analysis

The problem at its heart is naming a set of numerical values (the identifiers of the hidden services) in a way that is intrinsically meaningful to the average user. From this perspective the problem doesn't seem all that hard, as it already has been solved numerous times, with the *Domain Name System (DNS)* [2] probably being the most prominent solution.

Unfortunately, it is not that easy. Despite DNS being a distributed system, it still maintains a hierarchical structure, with the root (albeit consisting of multiple servers) being a central authority. In other words: should the operators of the authoritative root ever decide (or be coerced) to manipulate the system in any illegitimate way, there is little to defend against it, short of starting an alternative root. One example of such an attack was the "seizure" of numerous domain names in 2011 by the FBI [3], although the action was limited to the `.com` TLD instead of the whole DNS root [4].

2.1 Current Addressing Scheme

The current addressing system is described in the Tor Rendezvous Specification [5] and is rather simplistic. In order to set up a hidden service, the operator's OP has to generate a private/public pair of keys. The public key k is then combined with other pieces of information (like the later introduction points) to a data structure called *service descriptor*. There are two versions of service descriptors, V0 and V2, but we will ignore the former as it isn't supported anymore.

Among the data contained in a V2 service descriptor is a *permanent ID* that can be computed by calculating the SHA-1 digest $H(k)$ of the public key and truncating it after 80 bits or 10 bytes:

$$\text{permanent ID} = H(k)[:10]$$

Now the Onion URL can be obtained trivially by appending the string "`.onion`" to the Base32 encoding of such a permanent ID. Truncating $H(k)$ after 80 bits implies that half of the hash's output is being wasted, as SHA-1

produces a 160-bit digest. This is done to keep the URLs at a convenient length, currently at 16 alphanumeric characters (plus 6 characters for the pseudo-TLD). The permanent ID is also called service ID in the implementation, where it is used in both the "raw" and the Base32 encoded form.

Furthermore a service descriptor contains a *descriptor ID* that is calculated by concatenating the permanent ID with a *secret ID part* and hashing the result. The secret ID part in turn is calculated by hashing the concatenation of the *time period* (a periodically changing function of time and permanent ID), an optional *descriptor cookie* that is only shared with trusted clients, and a *replica* number:

$$\begin{aligned}
\text{descriptor ID} &= H(\text{permanent ID}|\text{secret ID part}) \\
&= H(\text{permanent ID}|H(\text{time period}|\text{descriptor cookie}|\text{replica})) \\
&= H(H(k)[:10]|H(\text{time period}|\text{descriptor cookie}|\text{replica}))
\end{aligned}$$

We aren't concerned about the mechanics of the secret key here. The important thing is that an OP can get to the descriptor ID by knowing just the permanent ID alone (and possibly the descriptor cookie, if the hidden service operator wants to limit the visitors to a certain group of trusted users).

A V2 service descriptor is finally *advertised* by signing and uploading it to a changing subset of 6 ORs (the DHT) that are chosen depending on their IDs and the descriptor ID of the record to be uploaded.

A user wanting to access a hidden service simply enters the URL received out-of-band. The user's application hands the URL to the OP, which in turn calculates the permanent ID and from that the descriptor ID. These can then be used to download and verify the whole service descriptor. From there the OP can build a circuit to one of the introduction points and everything that follows (see 1.1.1).

2.2 Zooko's Triangle

The first impulse now would be to devise an alternative naming system to DNS that meets all of Tor's needs. Unfortunately there is a conjecture that suggests our undertaking would be fruitless: *Zooko's Triangle* [6] [7].

It basically states its assertion in the title of the presenting article: "Names: *Decentralized*, *Secure*, *Human-Meaningful*: Choose Two". These three attributes are exactly the ones we would like to have for our naming system:

decentralized in the sense that there are no central authorities,

secure in the sense that an attacker cannot hijack a name,

human-meaningful in the sense that an average human can pick it up from the side of a bus driving by [7] [8].

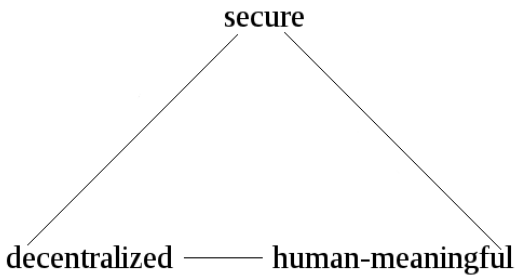

Figure 6: Zooko's Triangle

According to Zooko's Triangle *self-authenticating* names like the currently used Onion URLs cannot be meaningful to humans, where self-authenticating means that the correct mapping to a name is verifiable with the name (and only the name!) itself. They can, however, be secure and decentralized. *Non-self-authenticating* names like the ones used in regular DNS, on the other hand, cannot span trust boundaries. That basically boils down to: they cannot be used in a secure manner by peers independently of a third party. Therefore they are either decentralized and insecure or centralized and secure (or neither). Another example for non-self-authenticating names would be nicknames in IRC [9], as there is no way to stop someone from using

ones nickname in other networks, whereas looking at only a single network would violate the premise of decentralization.

2.3 Finding a Middle Ground

Zooko's triangle has shown that finding a "perfect" solution is improbable, but it certainly doesn't say that we cannot find a middle ground on the desired attributes. Transitioning from secure and decentralized Onion URLs to a scheme that compromises on these two attributes to gain a little human-readability may give some people an uneasy feeling, as it implies a reduction of security for a gain of usability, but that reduction may be of purely theoretical nature, as we will see.

2.3.1 Petnames

The first alternative to be considered is a so called *petname system*, as it is featured prominently in Tor's hidden service documentation.[6] The concept of petnames has initially been described in an article by Marc Stiegler that contains an extended interpretation of Zooko's triangle [8]. In this alternative rendering the author adds the concept of keys, nicknames, and petnames, each of which embody at most two of the three attributes in Zooko's triangle:

keys are decentralized, secure and never human meaningful,

nicknames are decentralized, human-meaningful and never secure,

petnames are secure, human-meaningful and never decentralized.

In the proposed system, each petname has to provide a bidirectional one-to-one mapping to a key, whereas a nickname only provides a mere one-to-many mapping to a key. The role of the "central" authority that manages the mapping between petnames and keys would then be assumed by the respective user of the system.

An example implementation of a petname system could be used to defend against phishing. For the sake of the argument assume a DNS domain name

[6] `https://www.torproject.org/docs/hidden-services.html.en`

to be some sort of nickname. The insecure aspect of domain names would be their vulnerability to mimicry (as in `examp1e.com`). If Alice now receives a link to Bob's website out-of-band, she has to decide whether to trust the link each time she sees it. Phishers are taking advantage of Alice's obligation to stay alert at all times.

Assuming the use of petnames Alice would only need to trust Bob once. The mapping between petname and key (e.g. "Bob's site" $\leftrightarrow$ `example.com`) is then saved and Alice could navigate from petname to domain name and the other way around in the future. If she ever follows a link to a bogus domain name instead of Bob's, she would (hopefully) notice that the site she landed on is not associated to the petname she expected to see.

The bookmark system that is currently used by most web browsers is related but weaker than the proposed petname system: a bookmark is not a bidirectional mapping, it can only be navigated in one direction (e.g. "Bob's site" $\rightarrow$ `example.com`) and is therefore called *lambda name* instead of petname.

Stiegler also asserts that a petname system as a whole embodies all the attributes of Zooko's triangle. Such a system has a serious drawback, however: petnames are strictly non-global, or, in Marc Stiegler's words: "The security properties of a petname come from its privacy." [8]

Another issue is that of application dependence. While it would be convenient to modify an existing petname plug-in for the Firefox browser, this would only be of advantage for a tiny set of protocols. Tor on the other hand aims to be an application agnostic tool for all TCP streams [10].

2.3.2 Word Lists

In search for a better solution Paul Crowley has written a two-parts essay that illustrate the use of word lists for a naming system [11]. There the author argues that Zooko's Triangle is actually a tetrahedron with four attributes instead of three, with three of four being achievable, the fourth attribute being:

transferable in the sense that the name can be transferred across a trust

boundary while preserving its meaning.

This property is assumed to be so central to most authors that it is not even mentioned most of the time (as for example in Zooko's original article). One notable exception is Stiegler's petname system (described above): such a system attains all attributes from Zooko's triangle but has to give up on transferability instead, with the consequence that petnames cannot be shared.

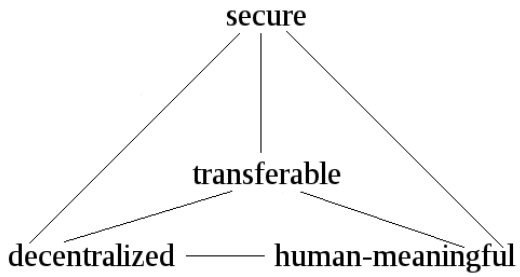

Figure 7: Zooko's Triangle with a fourth attribute in the third dimension

This alternative proposition aims for all properties, on the other hand. The underlying address mode of the system builds on distributed and secure keys, not unlike the currently used permanent IDs. Instead of exposing the user to a cryptographic key, however, each key k is mapped to a list of words from a given dictionary.

In the original essay this is achieved by hashing the key k and grouping the bits of the hash $H(k)$ in such a way that the resulting numbers can be used as indices to words in a predefined (and globally shared) dictionary. A 156-bit hash (as a general example with no direct relation to Tor) could be mapped to 12 words out of a dictionary of 8192 words in total. Domain names consisting of 12 words are still a far shot from being usable, but friendlier than the "naked" key at least (Fig. 8).

The author further elaborates on forcing the creator of a name to devote computational power on solving a challenge, a technique commonly known as *key stretching*. An example for such a challenge would be to find a 32-bit nonce n with a resulting hash $H(k, n)$ that ends in 23 zeroes. The first 65 bits of the hash could then be grouped into units of 13 bits that index 5 words

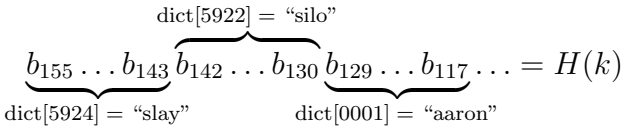

Figure 8: Barely memorable names with 3 of 12 words shown.

in a dictionary just like before. An attacker that wants to hijack a specific domain would still need to find one specific value in 2^{88}. The downside of this approach is that the creation of a name takes a few seconds of brute forcing on a typical computer, whereas lookup and validation should happen instantaneously.

Another enhancement mitigates birthday-attacks that are targeting a large number of names (e.g. all of them, in the case of vandalism). The problem is that the probability for a successful attack on an arbitrary name out of many could be substantially higher than the probability presented above. The proposed solution would be to include a handle h to the hash $H(h, k, n)$ that is used in the same manner as above. The handler could be chosen freely (within reasonable bounds, like domain names in DNS). Large scale attacks could be prevented this way and furthermore the handler would act as a user friendly denomination for specific hidden services (Fig. 9).

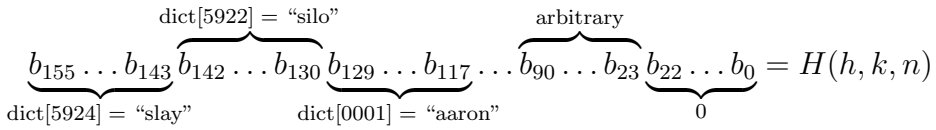

Figure 9: Simpler names with 3 of 5 words shown. The dictionary is the same as before.

One disadvantage of the word list approach in general is that the names still aren't as intuitive as the regular domain names. What has been dubbed human meaningful by Zooko is called readable or memorable by other authors; word lists surely qualify for being readable and somewhat memorable, but they are (arguably) meaningless.

2.3.3 Further Enhancements

Crowley's approach looks promising so far, but how does it fit into the existing picture? A URL of the proposed type would have to be resolved to provide the public key k and a nonce n, assuming the handle h is part of the URL received out-of-band. After that, the client could verify whether the returned information met all the constraints described above (namely that $H(h, k, n)$ could be broken up into the word list in the URL and that it ends in a certain predefined number of zeroes).

Unfortunately the already implemented lookup mechanism in Tor uses descriptor IDs (though these can be derived from the permanent ID which in turn can be derived from the public key k). Now there is no (known) way to derive the public key k from the hash $H(h, k, n)$, the same holds for the nonce n. We would have to reimplement the lookup mechanism of Tor, so the OP can look up k and n with $H(h, k, n)$ alone. After receiving this information, the OP would verify whether the answer was plausible and proceed by using the descriptor ID (derivable from the public key k) to download the service descriptor like in the old system.

Even if we could combine the first and second lookup in some way, the service descriptor would have to include the nonce n somewhere. Changing the format of the service descriptor (to V3) has to include a modification of the Tor Rendezvous Specification [5], a potentially complex and time-consuming process [12]. It would be much better if the changes necessary could be confined to the client side only.

A first step in that direction would be to eliminate the hashing of h, k and n. Since we are using a derivative of $H(k)$ for actually fetching the service descriptors, we might as well use (some of) the bits in that digest as the indices to words in the dictionary. One problem with this is that there are plans to increase the length of the public keys in the future.

Current proposals seem to be in favour of increasing the length of the permanent ID too (for example by taking the whole SHA-1 digest instead of the first half) [13]. Of course this would lead to word lists too long to be deemed acceptable. The problem could be prevented by leaving the definition

of the permanent ID as it is (i.e. at a length of 10 bytes) when increasing the length of the public keys.

The only part of the system that is getting weaker (or more precisely: not gaining strength) would be the part where the client validates whether it received the correct service descriptor. Fortunately we have already seen what to do to make brute-force attacks against the word list URLs harder: key stretching. Securing old-style URLs is not an issue, the client can always ensure it received a legit descriptor by checking whether the received public key matches the permanent ID in the entered URL.

The other problem is the nonce: it is unfeasible to include it in the service descriptor (as this requires the introduction of a new descriptor version in the Rendezvous Specification [5]). Therefore we would have to include it in the URL to be transmitted out-of-band. On the other hand, that would make the URL longer than desirable. It would be neater if the client could validate the key stretching step without any external factors, other than the URL.

We could just leave the nonce out of the game entirely, and only require the hash $H(h, k)$ to end in a certain number of zeroes. When an operator generates a URL, instead of varying a nonce, the OP varies the public key. Of course generating a key is much slower than varying a nonce, but we are in the lucky position to decide the number of bits we require to be zero.

Still, (ab)using the public key itself as the nonce is a risky endeavor. For one, existing implementations are never optimized for the creation of public keys. This means one would never know the orders of magnitude attackers are faster in creating keys than the regular users with their unoptimized key generators. Deciding on the "right" number of required zeroes will be a hard task, as a number too high would hamper the legitimate users, whereas a number too low would increase the attack surface.

One enhancement to Crowley's original proposal has been brought forward by Nick Mathewson in private correspondence (see appendix). In our current solution we would take the position of a word in a fixed directory to represent a part of the hash $H(k)$. Or, in the other direction, a group of bits in the hash $H(k)$ are interpreted as an index in the directory. The disadvantage is that the dictionary has to be the same for all users and is

practically impossible to modify after deployment, since each entry after an added or deleted word would change its position and thus its encoding in this scheme.

Mathewson's idea was to take the first few bits of the hashes of arbitrary words instead of their position for the same purpose (Fig. 10). This way there is no need to manage a dictionary (and even then it would be a lot easier, because it can be modified without changing the encoding of the words). Another advantage is that it is not language specific. The users could easily generate URLs in their own native language. One disadvantage would be that accepting arbitrary words could make them easily mimicked (for example by swapping characters in the middle of the word, a mistake easily overlooked by the human eye). Software to create deceptive imitations for SSH public keys already exists [14].

$$\underbrace{b_{64}\ldots b_{52}}_{\text{dict}[72c1_{16}]\,=\,\text{“shells”}}\overbrace{b_{51}\ldots b_{39}}^{\text{dict}[d022_{16}]\,=\,\text{“grassy”}}\underbrace{b_{38}\ldots b_{26}}_{\text{dict}[e158_{16}]\,=\,\text{“rot”}}\overbrace{b_{25}\ldots b_{13}}^{\text{dict}[9815_{16}]\,=\,\text{“fives”}}\underbrace{b_{12}\ldots b_{0}}_{\text{dict}[00e1_{16}]\,=\,\text{“pardon”}} = H(k)$$

$$\text{under the condition that } \overbrace{b_{64}\ldots b_{23}}^{\text{arbitrary}}\underbrace{b_{22}\ldots b_{0}}_{0} = H(h,k)$$

Figure 10: The final proposal. Note that the dictionary is not sorted alphabetically like before, but sorted with respect to the first 2 bytes of each word's hash digest.

A middle ground between a direct index and the hash approach could prove to be the most practical: There could be a dictionary (for different languages nonetheless), but the hashes of these words would be mapped to the groups of bits. That way we would retain the meaning of a word list even if the dictionary is changed in some way (e.g. by removing or inserting words) while preventing most name mimicry at the same time.

An experiment carried out by Mathewson (see appendix) suggests that over 98 percent of all 256-bit keys are representable with 16 English words (i.e. taking the first two bytes from the beginning of the hash). If the

generated key fails to produce any encoding, we could simply regenerate it, as we are already doing at the key stretching step presented above.

2.3.4 Nakanames

The most recent potential solution was proposed by Aaron Swartz in an essay from 2011 [15], the term *Nakanames* has been introduced in the FAQ to the essay [16]. It is named after Satoshi Nakamoto, the inventor of *Bitcoin*,[7] a virtual currency system that makes use of a very similar method.

The basic idea is to have a *scroll* S that can only be appended to. The scroll would consist of tuples (h, k, n) containing a name h, the associated key k and a nonce n. The whole scroll is then hashed from the beginning to the end; for the scroll to be valid, a certain number m of bits at the beginning of the hash $H(S)$ would have to be zero.

The scroll S would be distributed to a number of directory servers, in Tor's case either the directory authority or all the ORs. If there is more than one scroll, the longest is considered to be the authoritative one.

With this system, an operator would first download the authoritative scroll. Knowing that the authoritative scroll is valid, the operator would then try to find a nonce n such that the scroll S with an added tuple (h, k, n) still fulfills the requirement that there are m zeroes at the beginning of $H(S)$ with $(h, k, n) \in S$ where h is the operator's desired name and k is the key pointing to the associated service. The tuple is then distributed to the directory servers.

An attacker who wants to hijack a name would have to fabricate a scroll that is longer than the authoritative scroll. For the names in the scroll to remain valid, new nonces must be found for each tuple that has been appended since inclusion of the target. Depending on the age of the entry, this makes forging of domain names unpractical.

One big disadvantage of this approach is that names, once assigned, are immutable. If a hidden service loses a secret key for some reason, it cannot reclaim the domain after restoring operation. Of course this would be equally

[7] `http://www.bitcoin.org/`

true with the former propositions (but not with DNS) where this caveat is obvious, however. Then there are issues with preventing vandalism: a primitive DoS attack on this system would involve the creation of a large number of names. The scroll would grow dramatically and could never be shrunk back to a manageable size without collaboration from each and every directory server.

There are other criticism of the system [17] that lead to a discussion with the original author, where he states that his essay is to be seen as a proof of concept, not a specification for an implementation. A similar sentiment is evident in the FAQ where he states (on the question of downloading a 16 GB scroll on lookup): "By the time this system gets implemented, I suspect people will have plenty of bandwidth and space." [16]

We won't elaborate further on the criticisms mentioned above, as the assessment of the original author outweighs the advantages of the concept: there is much more research required to evaluate the real-world security and Tor is not the place to do this, as reaffirmed in Tor's design goals: "adding unproven techniques to the design threatens deployability, readability, and ease of security analysis." [10]

Namecoin[8] is an experimental name system based on Bitcoin and similar to Nakanames, albeit not targeted at Tor hidden services.

2.4 Conclusion

All in all Crowley's word list approach seems to be the safest bet for the time being. This decision has been reconfirmed by the Tor developers in private correspondence. Nakanames look promising too, but require substantial research before they can be put to the test responsibly in a productive system. Petnames, in contrast, solve a problem other than the one we are concerned with here. This certainly doesn't mean that they don't have their place: now that the Tor developers have disclosed plans to produce a fork of Firefox [18], a dedicated petname plug-in could complement the work presented in this thesis.

[8] `https://github.com/vinced/namecoin`

3 Specification

In this chapter some principal design decisions are outlined.

3.1 Use Cases

As the system to be implemented is mostly exposed to the users (both operators and mere "consumers", dubbed *visitors* below), we are in the lucky position to describe most desired properties with use cases.

There is one important aspect that cannot be described properly by a use case, though: the system has to be easily modifiable to accommodate defenses against future brute force attacks that are made possible by the steady increase in computational power.

3.1.1 Calling a hidden service with a memorable URL

This is the most obvious use case, as it is the central point of the whole thesis.

Precondition: the actor received a URL out-of-band
Postcondition: none

---

1. The visitor enters a memorable URL in an application associated with Tor
2. The derived descriptor ID is used to obtain the service descriptor from selected ORs
3. The OP checks whether the digest of handle and public key meets all the constraints explained above
4. The OP connects to the hidden service just like in the current implementation

3.1.2 Calling a hidden service with an old-fashioned URL

Despite the availability of memorable Onion URLs, some users may wish to continue using the traditional ones. Reasons might include legacy applications and existing bookmarks.

Precondition: the actor received a URL out-of-band
Postcondition: none

---

1. The visitor enters an old-fashioned URL in an application associated with Tor
2. The derived descriptor ID is used to obtain the service descriptor from selected ORs
3. The OP connects to the hidden service just like in the current implementation

3.1.3 Generating a memorable URL to a hidden service

Both operator and visitor could generate a memorable URL to a hidden service. Normally, the operator would do this after he generated the key pair for the languages he cares about. After finding a pair generating an appropriate word list, the resulting URL can be spread out-of-band.

Of course a visitor is free to generate another URL for an arbitrary hidden service, for example if the operator did not care about the user's native language. A URL like `example.fluid-area-above-movie-start.onion` could be "localized" to `example.metall-gelb-frieden-helm-zeit.onion`.

Of course there may be unpropitious adresses that don't allow localizations into a specific language, because they contain at least one of the few bit sequences that aren't associated with any word in the local dictionary. In the worst case the user would have to "loan" a foreign word from the original address, possibly after adding it to the local dictionary.

Precondition: the public key of the hidden service in question is known to the actor
Postcondition: a URL meeting the constraints explained above is known to the actor

---

1. The actor chooses a handler
2. The actor generates a URL from his local dictionary, such that all the constraints explained above are met

3.1.4 Setting up a new hidden service in the Tor network

Of course setting up a new hidden service has been implemented already. We have to make the procedure more computationally intensive, however, to mitigate brute-force attacks on memorable URLs. Also, we are amending the possibility to prefix the word list by a handler and introduce the possibility to recreate the keys in case the operator doesn't like the generated word list.

Precondition: none
Postcondition: none

---

1. The operator configures a hidden service
2. The OP generates a private/public key pair
3. If the operator specified a handler in the configuration file, generate a memorable URL for it
4. The operator has to either accept or reject the URL
5. If the operator is satisfied with the URL, the hidden service is advertised on selected ORs just like in the current system

The use cases can be summarized in a diagram:

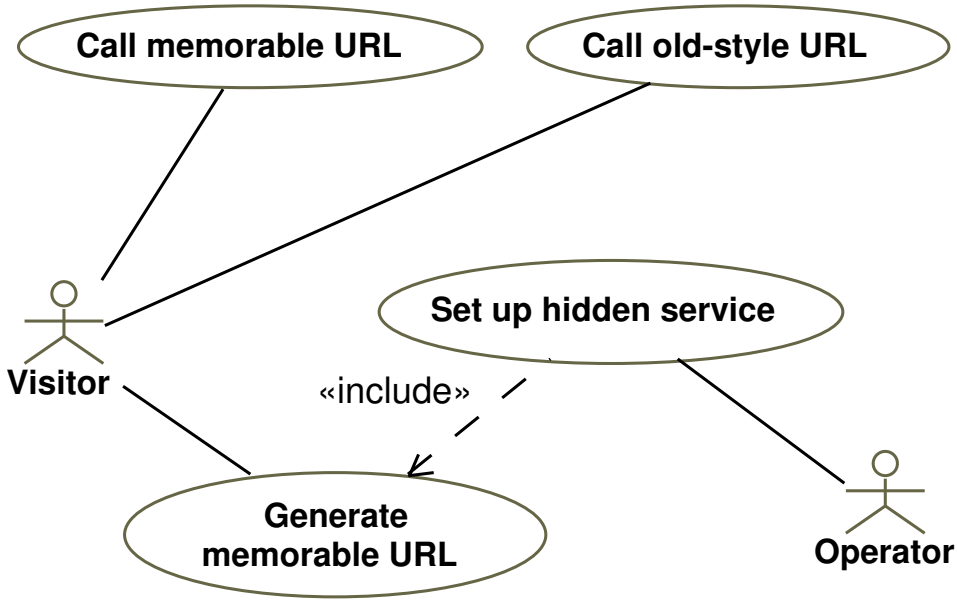

Figure 11: The UML diagram summarizing the above use cases.

To outline the next steps: we are now going to implement memorable URLs similar to the way suggested by Crowley [11]. The key difference is that the public key is used as the nonce, whereas in the original essay it has been a distinct object. Furthermore Mathewson's idea (see appendix) is implemented to make the dictionary optional and support different languages.

4 Implementation

Tor (or more precisely: its reference implementation) is mostly written in C. The part of the source code we are interested in is to be found in the directory `src/or/` (relative to the root of the Tor distribution). All statements concerning file names below refer to that directory, unless noted otherwise. The following descriptions have been deduced by reading the source code and are prone to change in the future.

4.1 Current Implementation

First we will take a look at the parts of the current implementation that are interesting for the problem at hand. We start our journey at the function `connection_ap_handshake_rewrite_and_attach` in `connection_edge.c`, as all clients calling an address will end up there. The function then decides what type the address belongs to by calling `parse_extended_hostname` from the same file. Possible types are:

`ONION_HOSTNAME` if the address is of the form "`y.onion`",

`EXIT_HOSTNAME` if the address is of the form "`y.exit`",

`NORMAL_HOSTNAME` otherwise (i.e. it isn't part of the Tor network).

Only the first type is of concern to us. Such an address would either be looked up in the cache with `rend_cache_lookup_entry` or, failing that, handed to `rend_client_refetch_v2_renddesc` in the file `rendclient.c`. Here things start to get interesting, as this function performs the actual lookup by calculating the descriptor ID with `rend_compute_v2_desc_id` and fetching the associated data from appointed ORs in the network by calling `directory_get_from_hs_dir`. The task of actually fetching the descriptor is delegated to `directory_initiate_command_routerstatus_rend` in the file `directory.c`. From there on things become too low-level to be of interest to us.

We turn our attention to the file `rendcommon.c` instead, where another function `rend_cache_store_v2_desc_as_client` is making sure that all of

the current constraints for service descriptors are met: the public key in the descriptor has to be encodable to the entered URL, for example. The age is also inspected; the descriptor must not have a time stamp too far in the past (up to 24 hours) or in the future (up to 1 hour) as per the Tor Rendezvous Specification [5].

The creation of URLs, on the other hand, is mostly done by the functions defined in `rendservice.c`. After the configuration file has been parsed by the functions defined in `config.c`, the function `rend_config_service` is called. This function sets up a structure `rend_service_t` that contains the properties of the hidden service.

In the function `options_act` in `config.c`, `rend_service_load_keys` is called for each hidden service specified in the configuration file. That function in turn generates a new key or loads an existing one from the file `private_key` in the hidden service directory. The permanent ID is also generated there and then written to the file `hostname` as a Onion URL.

That is all we need to know to add the new functionality to Tor.

4.2 Unit Tests

Instead of setting off to implement the feature immediately, unit tests have been written at each step (where it makes sense) in an effort to reap the benefits of *test-driven development*. Fortunately there already is a substantial base of unit tests (to be found in `src/test/`) that acts as a helpful point of orientation for newcomers.

After implementing the basic lookup functionality, however, additions to the unit tests became rare, as most of the later work either depends on interactive elements or a connection to the Tor network or requires individual files to be present on the machine. The lack of automated testing had to be made up by extensive and error-prone manual testing.

4.3 Basic Lookup Functionality

The basic lookup functionality consists of validating the entered URL, decoding it to a permanent ID and reencoding the permanent ID to an old-style

URL.

The first part has been achieved (after adding the relevant unit tests to `src/test/test.c`, which won't be mentioned from now on) by modifying the function `parse_extended_hostname` to make it accept the new-style Onion URLs. This required the introduction of a new function that has been dubbed `rend_valid_word_list_address` as an analog to the already existing function `rend_valid_service_id` that is called for old-style Onion URLs.

It has been decided that the new-style URLs shall bear the suffix `wl.onion`. That way we have a clear separation between the old and the new name spaces. Furthermore the SLD `wl` (for "word list") is interchangeable, should future developments of Tor lead to yet another name system; should Naka-names be implemented eventually, they could look like `example.naka.onion`.

The order of handle, word list, `wl` and `onion` was chosen to avoid confusion amongst users accommodated to DNS. Changing the position of handle and word list, for example, could lead to the fallacy that two different hidden services with the same handle are under administrative control of a common authority. Constraints of DNS have been implemented accordingly; there is a limit for the maximal length of a label at 63 octets and another for the whole domain name at 255 octets. The characters of words and handle are required to be alphanumeric characters for the time being, even though there isn't any technical reason why they couldn't be internationalized.

The next step was the introduction of `rend_decode_word_list` that decodes a word list in the manner described in the analysis above. This function optionally takes a dictionary as an argument. If specified, the words in the URL are searched for in this list. The returned data from the function is then used as an Onion URL just like in the original implementation.

Tor stores the Onion URL in a structure `struct rend_data_t`. We supplement it with the handle from the entered URL, because we will need it later (see 4.5).

4.4 Creation of URLs

The first decision to make is how creation of a word list URL has to be communicated by a hidden service operator. As the handle is required with word list URLs and it is senseless in the context of old-style URLs, it was decided not to introduce a distinct option for word list URLs. The handle option `HiddenServiceHandle` implicitly activates the creation of these addresses, instead. As the new-style URLs are supposed to be complementing the old addresses, old-style URLs are generated in any case and can be used just like they were before.

Other options have been introduced too: `ConsultDictionaryOnResolve`, `GenerateWordListOnResolve` as well as `DictionaryFile`. The first value is a boolean that controls whether all the words in a word list URL are to be confirmed to exist in the dictionary. This should protect against most URL mimicry with (intentionally placed) typos. The second options is a boolean value too. That way the user can control whether Tor shall generate (and print) new word list addresses for each hidden service visited.

The option `HiddenServiceHandle` is local, it has to be specified for each hidden service the operator wishes to generate a word list URL for. Generating such a URL or consulting the dictionary on resolve of course both require `DictionaryFile` to be set. The other options are global.

They where easily added by inserting them into the array `_option_vars` in the file `config.c`. The default values of `ConsultDictionaryOnResolve` and `GenerateWordListOnResolve` are false, because loading the directory is an expensive operation, as we will see.

We also wish to output the chance for a word list encoding to exist when the dictionary has been loaded afresh. Basic probability calculus helps us calculating the chances for 5 values (of two byte each) to match the beginnings of the SHA-1 hashes of any word in a set with n words in total. In other words: given a random service ID, the probability for a word list encoding to exist is

$$P(\text{“word list encoding”}) = \left(\frac{n}{2^{16}}\right)^5 = \frac{n^5}{2^{80}}$$

This is much lower than perhaps assumed intuitively. Let us consider a fairly complete dictionary with 88105 words, extracted from one of the included dictionaries in *GNU Aspell*.[9] These words are producing 48015 different two byte hashes. The probability for a word list encoding (with 5 words) to exist is now only 21.11 percent! The key stretching described below lowers the probability for a usable encoding to exist even more.

As loading the dictionary file is the most resource intensive operation, typically taking around two minutes, we want to avoid it whenever it is possible. A number of ideas have been implemented to help with this problem:

- The dictionary is only accessible through an interface that caches all the words in the current dictionary.
- The actual loading is deferred for as long as possible. That way a user, even if `ConsultDictionaryOnResolve` has been specified, doesn't have to load the dictionary, as long as no word list URLs are entered.
- When caching the dictionary, the words are stored in appropriate, efficient data structures.

To check whether words are in the dictionary, we append all of them to a list named `dictlist`. To generate a word list URL, on the other hand, we only need the shortest words for each particular two byte value. Therefore we used a *digest map* named `dictmap`, that is, a data structure that maps the digests of words to the words themselves.

With both structures only the references to words are used, so there is hardly any memory overhead despite seemingly duplicated storing. Care is warranted when freeing data structures, however.

While the dictionary is loading, the Tor process is blocked. One could avoid that by moving the loading part to another thread, but fortunately Tor is robust enough to deal with the blocking typically seen in our use case. The worst effect experienced was dropping of old circuits that had to be rebuilt.

When consulting the dictionary on resolving an address, we look up the words in the URL in `dictlist`. Here we could reach a complexity of

[9] `http://aspell.net/`

$\mathcal{O}(\log(n))$ if we would sort the list prior use. On the other hand, the delay for searching the words has not been noticeable in tests so far.

The function `rend_encode_word_list` is the counterpart to the decode function presented before. One important difference is that the dictionary is not optional here. Where should the words in the result come from, otherwise? This is also the reason why `ConsultDictionaryOnCreate` is missing.

If the function returns successfully, we know we have an encoding and can upload the service descriptor just like in the original implementation. If there is no valid word list encoding, however, we have to generate a different public key. The function `init_key_from_file` from the original implementation has been modified for that reason, such that it overwrites an existing key file on request.

We have to be careful here, as not to overwrite older key files by accident. If Tor encounters a pre-existing key file in a hidden service directory, it will always leave it alone. The operator has to delete the file by hand if a new URL is desired. Similarly, Tor will only regenerate a URL if there is no `hostname` file in the hidden service directory.

4.5 Key Stretching

Implementing the key stretching was rather straightforward. The function `rend_digest_ends_in_nuls` is called both when generating and when resolving a URL. Its arguments are the handle, the public key's digest and the number of NUL-bytes the resulting digest has to end in for the function to return true. As the third argument would typically be a single constant in the program, one may wonder why it is specifiable here. We will soon see the reason for this, but let us consider the problems first.

The one big disadvantage with this approach is that it is nearly impossible to seamlessly increment the number of required zeroes after the system has been deployed, as then all hidden services would have to regenerate their public keys, thus changing their URLs (the word list part, at least).

On the other hand, what can be done? Preserving the public key would require some other variable the hidden services can alter and upload to the

DHT, where a client could then download that variable and test whether the key stretching took place. The only clean way to achieve this would be to introduce a nonce (or a similar piece of data) and that in turn would require a change of the rendezvous specification. We are back at Crowley's original idea now.

An alternative form of key stretching might buy us some time: we could apply the hash function repeatedly (many thousand times) in a nested fashion before checking for trailing zeroes. This would slow down both generation and resolving of word list URLs. Unfortunately it would have exactly the same problems as the current solution.

In the end a compromise has been implemented that hopefully makes a transition to a more complex challenge a possibility, at least. Two different constants for the number of required zeroes have been defined:

`REND_WORD_LIST_ACCEPT_NNULS` for the required zeroes if resolving a URL,

`REND_WORD_LIST_GENERATE_NNULS` for the required zeroes if creating a URL.

The former is supposed to be less or equal the latter. That way the Tor developers can open a transition window for the hidden services by specifying different values for these constants. That is the reason the function `rend_digest_ends_in_nuls` takes the required number of zeroes, as it depends on the context the function is called in.

Operators would generate a new URL for their hidden service in such a case. Then they could advertise their new URL at the old addresses that are still accepted by clients for the length of the transition window. By the time the transition window is closed, hopefully all users know about the new locations. While certainly an imperfect solution, it is probably the best we can do without changing the rendezvous specification [5].

4.6 Evaluation

The result of the work described above is a patch of around 1000 lines of code. The changes are as non-invasive as possible, for there are less chances for things to go wrong. Faced with the choice between changing the content

of an existing array in-place or creating a working copy of it, the latter was almost always the preferred way. It might be less efficient, but we can be sure that existing facilities won't get confused by our changes.

Much attention has been devoted to secure coding principles, for example we always used *OpenBSD*'s[10] `strlcpy` and `strlcat` [19] instead of the ANSI counterparts as well a Tor variants of other standard functions (e.g. `tor_strdup`). The changed code compiles with GCC's warnings enabled and has been tested for memory leaks, double frees, and the like with *Valgrind*.[11]

Care has been taken to ensure that our modifications are portable across a wide range of different operating systems and architectures. Tor should run at least on Windows, Linux, Mac OS, BSD and Solaris. [10]

[10] `http://www.openbsd.org/`
[11] `http://valgrind.org/`

5 Conclusion

The solution presented in this thesis should make hidden services more accessible to a wider audience and furthermore have an everlasting, positive impact on the security for psychological reasons. Word list addresses in their current form are more secure even from a theoretical viewpoint than the old-style addresses, thanks to key stretching. The problems begin with the introduction of longer public keys at some point in the future.

As we have seen, future changes may bring an end to word list URLs. Unfortunately it turned out that addressing the issue won't be possible without changing at least the rendezvous specification [5] (and possibly the Tor specification too), as there is no satisfying way for a client to validate that the required key stretching step took place at a hidden service. Using the public key's digest plus handle for this is a less-than-ideal solution. A future revision of the specification could include a nonce as envisioned in the original essay, for example.

There are other proposals to ensure that proper efforts were invested in creating a word-list URL. Some of the proposals [20] have the advantage that they are memory-bound rather than CPU-bound. This is important because the contrast between low and high end systems is much less daunting in this regard. Ideally changes to the specification to include the required data in the service descriptors should be made concurrently with the introduction of longer keys.

Another limitation is the handling of the dictionary, albeit it is dealt with much more easily than the previous issue. There has been nearly no performance tuning on the loading process, other than a few basic design decisions. We cannot go without loading the file one way or another, but there is plenty of room for improvement. The dictionary could be loaded in a separate thread while Tor is bootstrapping, for example. Another enhancement could be made by introducing a binary file that acts as a cache between program starts.

Yet another remaining problem became evident while testing the basic functionality with a web browser: some of the hidden services failed to re-

spond to our queries with the expected page. The explanation for this behaviour lies in the specification of HTTP 1.1 [21]. A browser has to submit the hostname with each HTTP GET request, such that a web server with virtual hosts knows which page to return. In our case the browser sends the URL entered by the user whereas many hidden services are expecting the old-style Onion URL.

Unfortunately none of the solutions to this problem are appealing. The simplest solution would be to wait until the operators configure their web servers to ignore the hostname sent in HTTP. That way the ability to host multiple sites on a single server could be constricted. On the other hand this is arguably a bad idea to begin with, since an attacker could identify all websites by subverting a single server. The other problem is that the operators are able to see which URL a visitor used in a request. They could take advantage of this information to partition their visitors.

The alternative solution would be to include a mechanism so that the browser encodes the entered URL to an old-style URL before sending it. This would be a substantial duplication of the work done in Tor itself. Moving the whole functionality to the browser is unacceptable either, as we would lose it in other (non-web) applications then.

A Nick Mathewsons Email

Thanks to Nick Mathewson for allowing me to publish this private mail:

```
On Mon, Feb 21, 2011 at 9:10 PM, Simon Nicolussi
<Simon.Nicolussi@student.uibk.ac.at> wrote:
 [...]
> Paul Crowley used a list from Diceware without any words that contained
> special symbols (http://world.std.com/~reinhold/diceware.wordlist.asc).
> I'm not sure about _this_ list, but I'm pretty confident that there are
> enough distinct words in English to make a list that easily matches our
> requirements. Given enough distinct words, one could even build 5-word
> sentences instead of simple lists. This would enhance the memorability
> of the URLs even more.

If we wanted, we could make the wordlist optional on the client side
and have it be required only on the hidden service side.  The trick is
to define the client-side name->bits mapping for a human-readable name
as something like:
   "subname1.subname2....subnameN" ->
       H(subname1)[:nb] | H(subname2)[:nb] | ... | H(subnameN)[:nb]
where H is a hash function, x[:nb] is the first nb bits of x, and | is
concatenation.

For nb == 16, this means we need 5 words per 80-bit key, 10 per
160-bit key (if we go there in the future) and 16 per 256-bit key (if
we go _there_ in the future).  Note that since the hash function _is_
the word mapping, we don't actually need a fixed word list on
client-side, and servers can use whatever word lists they want.  That
could be advantageous if we want to allow different languages!

But this scheme depends on us having enough words so that, for any
nb-bit sequence, we have a tolerable word whose hash starts with that
sequence.  Are there enough words in English for that?  I'll leave the
combinatorics for your thesis and instead attach a trivial python
script that finds the shortest word in /usr/share/dict/words such that
the first 2 bytes of SHA1(w) is x for all two-byte x.  On my desktop's
English dictionary, it gets hits for 65465/65536 possible 16-bit
sequences: over  98% of all 256-bit keys are representable with 16
words.  [If you generated a key that fell in the bad 2%, you could
```

```
just generate a new one.]

For example, the key in your original message could become:
  "woodwose.temin.gerate.ungagged.swosh.onion" or
  "callean.usss.gerate.deniers.current.onion" or even
  "hair-raising.flippant.rontgens.deniers.nonspheral.onion"

Is this approach a win?  Some advantages here are:
  * we don't need to ship a wordlist for clients
  * we can support all languages, not just English.
  * It falls back reasonably well on the old compact names.

Some disadvantages are:
  * It's not certain that you'll find an encoding for an arbitrary string.
  * The words are less likely to be common ones.
  * There can be more than one encoding for a given string.
```

wordhash.py

```
import sha
import binascii
import re

def z(words):
  d = {}
  for w in words:
    w = w.strip().lower()
    if not re.match(r'^[a-z0-9\-]*$', w): continue
    s = sha.sha(w).digest()[:2]
    if d.has_key(s):
      if len(d[s]) > len(w):
        d[s] = w
        #print "%r -> %s"%(s,w)
    else:
      d[s] = w
      #print "%r -> %s"%(s,w)
  return d

for s,w in sorted(z(open("/usr/share/dict/words", 'r')).items()):
  print binascii.b2a_hex(s), w
```

References

[1] R. Dingledine and N. Mathewson, "Anonymity Loves Company: Usuability and the Network Effekt," in *Proceedings of WEIS 2006*, June 2006.

[2] P. Mockapetris, "Domain Names - Concepts and Facilities." available at `http://tools.ietf.org/html/rfc1034`, November 1987.

[3] E. Davies, C. Sullivan, J. Richardson, E. Rivera, T. Flanelly, and J. Margolin, "Manhattan U.S. attorney charges principals of three largest Internet poker companies with bank fraud, illegal gambling offenses and laundering billions in illegal gambling proceeds," April 2011.

[4] J. S. Bourne, "What it Means When U.S. Law Enforcement "Seizes" a Domain Name." available at `http://www.domainnamestrategy.com/2011/04/21/vocabulary-lesson-what-it-means-when-us-law-enforcement-seizes-domain-name`, April 2011.

[5] R. Ransom, R. Dingledine, N. Mathewson, K. Loesing, S. Hahn, and A. Lewman, "Tor Rendezvous Specification." available at `https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=rend-spec.txt`, June 2011.

[6] Z. Wilcox-O'Hearn, "Names: Decentralized, Secure, Human-Meaningful: Choose Two." available at `http://www.zooko.com/distnames.html`, September 2003.

[7] C. Shirky, "Domain Names: Memorable, Global, Non-political?." available at `http://shirky.com/writings/domain_names.html`, May 2002.

[8] M. Stiegler, "An Introduction to Petname Systems." available at `http://www.skyhunter.com/marcs/petnames/IntroPetNames.html`, June 2010.

[9] J. Oikarinen and D. Reed, "Internet Relay Chat Protocol." available at `http://tools.ietf.org/html/rfc1459`, May 1993.

[10] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The Second-Generation Onion Router," in *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[11] P. Crowley, "Squaring Zooko's triangle." available at `http://www.lshift.net/blog/2007/11/10/squaring-zookos-triangle` and `http://www.lshift.net/blog/2007/11/21/squaring-zookos-triangle-part-two`, November 2007.

[12] N. Mathewson, "The Tor Proposal Process." available at `https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=proposals/001-process.txt`, January 2007.

[13] N. Mathewson, "Initial thoughts on migrating Tor to new cryptography." available at `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-crypto-migration.txt`, December 2010.

[14] K. Rieck, "Fuzzy Fingerprints, Attacking Vulnerabilities in the Human Brain." available at `http://freeworld.thc.org/papers/ffp.pdf`, October 2003.

[15] A. Swartz, "Squaring the Triangle: Secure, Decentralized, Human-Readable Names." available at `http://www.aaronsw.com/weblog/squarezooko`, January 2011.

[16] A. Swartz, "Frequently Asked Questions." available at `https://squaretriangle.jottit.com/faq`, January 2011.

[17] D. Kaminsky, "Spelunking the Triangle: Exploring Aaron Swartz's Take On Zooko's Triangle." available at `http://dankaminsky.com/2011/01/13/spelunk-tri/`, January 2011.

[18] M. Perry, "To Toggle, or not to Toggle: The End of Torbutton." available at `https://blog.torproject.org/blog/toggle-or-not-toggle-end-torbutton`, May 2011.

[19] T. C. Miller and T. de Raadt, "strlcpy and strlcat - Consistent, Safe String Copy and Concatenation," in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 1999.

[20] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately Hard, Memory-bound Functions," *ACM Transactions on Internet Technology*, vol. 5, p. 299 to 327, May 2005.

[21] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1." available at `http://tools.ietf.org/html/rfc2616`, June 1999.