Improving the Privacy of Tor Onion Services*

Edward Eaton, Sajin Sasy, and Ian Goldberg

University of Waterloo, Waterloo, ON, Canada
{eeaton, ssasy, iang}@uwaterloo.ca

Abstract. Onion services enable bidirectional anonymity for parties that communicate over the Tor network, thus providing improved privacy properties compared to standard TLS connections. Since these services are designed to support server-side anonymity, the entry points for these services shuffle across the Tor network periodically. In order to connect to an onion service at a given time, the client has to resolve the `.onion` address for the service, which requires querying volunteer Tor nodes called Hidden Service Directories (HSDirs). However, previous work has shown that these nodes may be untrustworthy, and can learn or leak the metadata about which onion services are being accessed. In this paper, we present a new class of attacks that can be performed by malicious HSDirs against the current generation (v3) of onion services. These attacks target the *unlinkability* of onion services, allowing some services to be tracked over time.
To restore unlinkability, we propose a number of concrete designs that use Private Information Retrieval (PIR) to hide information about which service is being queried, even from the HSDirs themselves. We examine the three major classes of PIR schemes, and analyze their performance, security, and how they fit into Tor in this context. We provide and evaluate implementations and end-to-end integrations, and make concrete suggestions to show how these schemes could be used in Tor to minimize the negative impact on performance while providing the most security.

Keywords: Tor · Onion Services · Unlinkability · PIR

1 Introduction

Tor provides anonymity to millions of users accessing the Internet every day [31]. However, Tor can also be used to provide this same protection to *hosts* of content, resulting in bidirectional anonymity (or pseudonymity). This is achieved through the use of Tor *onion services*[1] [12]. Communication over Tor is done using Tor *circuits*. A circuit is a chain of typically three relay nodes through which traffic, encrypted in layers, is sent. The first node in a circuit is a client's

* This is an extended version of our paper that appeared in the proceedings of the 20th International Conference on Applied Cryptography and Network Security (ACNS 2022) [13].

[1] Onion services were originally called hidden services, and hence some of the related nomenclature still uses the word "hidden" instead of "onion" (for example, "HSDir").

guard node. In order to prevent certain attacks, a client will try to use the same guard node for every circuit it builds over the course of several months [11]. In order to communicate with an onion service, the client must learn the location of a Tor relay that has an open circuit to the onion service, called an *introduction point*. Each onion service typically has multiple introduction points distributed across the Tor network, which maintain circuits that connect to the onion service. Clients use these introduction points to inform the onion service of a *rendezvous point* that the onion service and client can communicate through via Tor circuits. In order to start this process the client must first obtain a list of the introduction points an onion service uses. This is done by querying the *hidden service directories*, or HSDirs. HSDirs assist the client in the task of translating the `.onion` address of an onion service into its list of introduction points. Onion addresses are encodings of the long-term identity public key owned by the onion service. For example, an onion address looks like: vww6ybal4bd7szmgncyruucpgfkqahzddi37ktceo3ah7ngmcopnpyyd.onion. How this address is distributed to users varies according to the onion service in question.

In version 3 of the onion services protocol, which we focus on in this paper, the onion address is used to query the HSDirs by first translating the original public key to a new 'blinded' public key, which changes at a regular interval (currently one day). The client uses the blinded public key to query the HSDirs, who provide the client with the *descriptor* associated with that blinded public key. These descriptors are encrypted under a symmetric key that can be derived from the onion address and contain a list of the introduction points. Each descriptor is held by a pseudorandom subset of Tor relays with the HSDir flag enabled. The mapping of which relays hold which descriptors changes over time, and is determined by a variety of inputs and system parameters, including the blinded public key of the onion service, the identity of the relay, and a shared random value distributed across the network; this shared randomness makes it hard for a malicious adversary that intends to censor an onion service to a priori compute and target the relays that will be used for serving the descriptors of an onion service in the future. We describe this process in detail in Section 2.1.

Tor has deprecated version 2 of the onion service protocol as of July 2021. An explicit goal of version 3 of the onion services protocol is that it should be difficult for the HSDirs to know i) which onion services they hold descriptors for, and ii) which onion services are being accessed when they are queried [35]. In version 2 of the protocol, the permanent public key associated with the onion service was contained within the descriptor, allowing HSDirs to identify the onion services they hold descriptors for. To protect against this, in version 3 the HSDirs hold descriptors indexed by a blinded public key, and since the identity public key (the onion address) cannot be recovered from the blinded key, the HSDirs cannot link an onion service descriptor with the underlying onion service unless it already knows the identity public key for the onion service. This process of keyblinding provides the security property of "unlinkability", which states that for an adversary who only observes blinded public keys and signatures under

those keys, it is cryptographically impossible to pair two blinded keys as having the same underlying identity key.

However, in many cases it is reasonable to expect that the HSDir may know the identity public key. For example, many onion services widely distribute their `.onion` address so that anyone can access them, or a malicious adversary trying to deanonymize an onion service may get the public keys in some other way. Any HSDir can check if they hold the descriptor for an onion service that they know the identity public key for, simply by deriving the blinded public key and checking the descriptors they hold. In this work we consider how an HSDir's ability to connect incoming descriptor queries to blinded public keys impacts the privacy of onion services in Tor. We find that the information HSDirs have access to puts them in an advantageous position for launching attacks that can harm the anonymity of both onion services and clients connecting to those services. In particular we find that for onion services that wish to remain unknown, but are relatively popular within a community, it may be possible to violate unlinkability. To prevent this source of information being available, we explore the integration of Private Information Retrieval (PIR) into the descriptor lookup process.

1.1 Related Work

The idea of using PIR to mask the relative popularity of onion service queries from (version 2) HSDirs was mentioned in a blog post by Kadianakis in 2013 [19]. This blog post outlined various deficiencies in the way onion services worked (at the time), as well as proposing research directions for the scientific community to investigate to address these problems. In particular, while the post suggested using PIR as a possible solution to this problem, it did not investigate what kinds of PIR would be ideal or propose how the PIR schemes would actually be integrated into Tor. Ours is the first work that characterizes attacks against the newer version 3 onion services, explores the design space of a PIR-based solution to address it, and provides an implementation to demonstrate the effectiveness of those designs.

Other integrations of PIR into Tor have been explored before [25, 29], but the focus in those works was on using PIR for finding nodes to build circuits, and not for onion service queries, although the two do share similar interests. Consideration of the possibility and the ramifications of malicious HSDirs has been addressed in some works before [17, 23, 28], but no work has yet explored whether knowledge of the distribution of queries made to an HSDir could be used to aid attacks that deanonymize clients or onion services.

Our Contributions.

1. We provide a description of the v3 onion service lookup process, which is key to the remainder of the text in Section 2. We then analyze the leakage induced by the descriptor lookup mechanism of v3 onion services, and propose attacks targeting both clients and onion services that leverage this leakage.

2. In Section 3 we discuss the variants of PIR that could solve this problem, and address the challenges of integrating them into a complete end-to-end solution for Tor's onion services, while also highlighting a network-level optimization that saves an additional network round-trip that PIR would otherwise introduce.
3. We analyze the different PIR schemes proposed to compare the privacy guarantees provided by each in the context of Tor, using enumerative techniques to provide an upper bound on the probability an adversary is able to compromise our multi-server PIR system in Section 4.
4. Finally, in Section 5 we provide microbenchmarks for all of the defences we propose. Additionally, we also implement and evaluate an end-to-end integration for the best solution from our microbenchmarks on the live Tor network. Our results demonstrate that the performance overhead (or effect on time to load an onion service for clients) of our proposed defence is negligible.

2 Attacks

In this section we describe various avenues of attack enabled by the metadata currently provided by query lookups. We start with an overview of the lookup process, so that we can motivate our adversarial model and explain the attacks. Next we consider two broad classes of attacks: those targeting clients, and those targeting hidden services. For each class, we argue how a malicious HSDir may leverage the information of the distribution of query lookups to gain information on a target or track them over time.

2.1 Tor and Hidden Service Directories

The Tor network is run by thousands of volunteer nodes, called *relays*. As of January 2022, there are roughly 7000 relays that forward traffic for the Tor network [31]. These relays are listed in the Tor *network consensus*, which is a document generated by the nine Tor network directory authorities once per hour [32]. This consensus lists some global *parameters* conveying information about how Tor clients and relays should behave, and each relay listed in the consensus can also have *flags* indicating what properties the relay has. Currently, roughly 4000 relays have the "HSDir" flag, indicating that the relay will hold descriptors for onion services and deliver them to clients. Time is divided into *epochs*, with the size of the epoch being a consensus parameter of the Tor network. Currently, the length of an epoch is one day.

The HSDirs collectively store the onion service descriptors in a distributed hash table. Each epoch, each node has a separate index value, denoted `hsdir_index`. These indices are unpredictable and uncontrollable by the HSDirs. By ordering these indices, and looping back at the end, we can form a ring of the HSDirs. For redundancy, each descriptor is held by multiple HSDirs. To determine which HSDirs hold which descriptors, we can calculate $\texttt{hs\_index}_i$ values for the onion service, where i ranges from 1 to `hsdir_n_replicas` (a parameter given in the

consensus, currently 2). These $\texttt{hs\_index}_i$ values are determined by the blinded public key of the onion service for that epoch, as well as the index i and a few consensus parameters. The descriptor is uploaded to the `hsdir_spread_store` (currently 4) HSDirs whose `hsdir_index` values come directly after the $\texttt{hs\_index}_i$ values in the HSDir ring. In this way, the descriptor is replicated across the hash ring multiple times (currently 8) in each epoch, so that a client wishing to access the descriptor has a variety of HSDirs that may be queried, improving the privacy and availability of the onion service.

The Tor metrics site [31] provides some sense of the current scale of onion service usage. It reports around 4000 HSDirs (Tor nodes with the HSDir flag) in the network. Since their deprecation, the number of v2 onion services has dwindled to around 25 thousand, while the number of v3 onion services has steadily increased since tracking began (in September 2021), to around 700 thousand today. Given the number of HSDirs, the number of unique services, and the number of times a descriptor is replicated, we arrive at a rough estimate of $700000 \times 8/4000 = 1400$ descriptors on average per HSDir.

For the remainder of this section, we want to consider the capabilities of a malicious entity willing to act as an HSDir. To model this, we consider that the Tor network has n relays with the HSDir flag. We envision that our adversary controls a of these relays. This adversary can see all incoming HS lookup queries on the HSDirs that they control. As we will see, even with this simple model, the adversary can draw conclusions and make inferences about both clients and onion services that go beyond what Tor allows for from other nodes in the network.

2.2 Attacks Targeting Clients

An adversary hoping to deanonymize a Tor user who uses onion services is placed in a relatively powerful position in the network. When a client resolves an onion service descriptor lookup, they connect to the HSDir via a circuit. Hence if an adversary in addition to controlling the HSDir, controls even the middle node of this circuit to the HSDir, they learn both the client's guard relay and the blinded public key of the service being connected to.[2]

For a service that widely distributes their `.onion` address, this gives the adversary the client's entry point to the network (the guard relay) and their final destination (the onion service). As clients maintain their guard node for a long period of time (currently up to six months) [33, 14], the guard relay itself provides substantial information that can allow a malicious actor to trace a client over time. Combined with the information of the final destination, this can lead to powerful epistemic attacks [8, 9].

[2] Of course if the adversary controls the guard relay, they have an even stronger attack vector, and will learn the client's true IP address. However it is easier for an adversary to control middle nodes, since attaining guard status for a relay requires uptime on the order of several weeks, and clients do not often select new guards.

2.3 Attacks Targeting Onion Services

The introduction of blinded public keys in version 3 of the onion services protocol intended to provide better anonymity properties for onion services against malicious HSDirs. Blinded public keys cannot be traced back to the identity public key, and the blinded public key changes with each epoch; therefore, in theory onion services cannot be tracked by HSDirs across epochs. Cryptographically, this is formulated as *unlinkability*, which states that after observing many public keys and many signatures under those public keys, an adversary cannot do better than guessing to link two blinded public keys as being derived from the same identity key. In this subsection, we argue that while the cryptography used for blinded key schemes is solid, these guarantees do not extend to all onion services in practice because of how descriptor lookups are resolved.

To track an onion service over time, an HSDir can consider the *distribution* of queries made to each service over time. Different services are likely to have radically different distributions of queries. By identifying two blinded public keys that received a similar distribution of queries over the course of an epoch, an adversary can ascertain with a reasonable degree of confidence that the two blinded keys correspond to the same identity public key. The challenge in this setting is that the database is distributed, and hence the adversary's view is limited to a fraction of the total set of queries made within an epoch. This fraction is defined by the adversarial power a and the total number of nodes n. This notion of building a 'profile' for an onion service based on the query distribution is simply the starting point of our attack, and we will refer to it as the *weak variant* of the attack. A truly malicious adversary has several other sources of information available to them that strengthens the profiles constructed. For a given onion service, additional sources of metadata that a malicious HSDir could leverage include (i) the set of guard nodes that make the HSDir lookup requests, (ii) the frequency distribution of lookup requests from the aforementioned set of guard nodes, or (iii) the timings of lookup requests within an epoch.

There are other information channels possibly available to an adversary as well, such as considering correlations between the timing of queries to cross-linked onion services. Nonetheless, the common underlying element that makes the attack feasible is *the ability of an adversarial HSDir to infer which of its onion service descriptors is being looked up in a request, allowing them to link the metadata of the request to that particular onion service.* The solutions that we propose in Section 3 will prevent these attacks we outlined.

3 PIR for Descriptor Lookups

To prevent the kinds of attacks established in Section 2, we need to prevent malicious HSDirs from learning which descriptor is being queried by a user. As a general approach, the obvious tool for this requirement is Private Information Retrieval (PIR). However, there is a large research gap between the simple idea of using PIR and its actual integration into the descriptor lookup process. We consider the three approaches of multi-server PIR, single-server PIR using

computational assumptions (CPIR), and single-server PIR using hardware assumptions. For the remainder of the paper, we show how these different PIR schemes can be integrated into Tor to prevent the statistical attacks we have shown. Single-server PIR does not change how clients decide which HSDir to query from. The structure of the hash ring, how the client decides where to query from, and the logic the onion service uses to decide where to upload can all stay the same, while multi-server PIR does introduce significant changes to this structure. We provide a complete system design for how to integrate each of these into Tor, and explore the advantages and disadvantages of each approach. Integrating a PIR scheme into any application context poses several challenges of bridging the rigid semantics of a PIR scheme with the underlying architecture:

1. Commonly, records in PIR schemes are stored as logical arrays and are indexed by an integer; in order to retrieve a particular record the client has to query for the corresponding index of this record. However in our context of onion service descriptors, the data to be queried privately is a key-value store, and we discuss later in this section how to bridge this gap.
2. PIR schemes are computationally expensive and hence in order to ensure that integrating PIR guarantees into queries does not impact the performance of the entire application we need to ensure that PIR queries are handled asynchronously. In Appendix A we give the technical details of how we extend Tor's program architecture to support asynchronous PIR queries on both the client and server side.
3. In order to even construct a PIR query, the client needs to a priori know the parameters of the PIR scheme it is interacting with; this will induce a performance penalty of an additional round trip of communication for learning those parameters. Note that the delay introduced by the additional round trip directly impacts the time for an onion service's page to start rendering, which is an important user-experience metric to minimize. We provide an optional optimization that can remove this additional round trip in Appendix B by leveraging the fact that these parameters can be publicly published.

Single-server PIR. In a single-server scheme, a client queries a database by sending an encrypted version of the index that they want to retrieve data for. The server performs some computation over the database, and returns an encrypted response without learning any information about the index. This goal can be accomplished either by using encryption with strong mathematical properties like fully homomorphic encryption, as in XPIR [1] or SealPIR [2], or by using secure hardware, as in ZeroTrace [30].

Multi-server PIR. Multi-server schemes instead have the database distributed across several servers. The client must query these servers to obtain some data that can be recombined to obtain the query result. These schemes rely on non-collusion assumptions; if the queried servers collude with each other, they can determine what a client has queried. In a multi-server PIR scheme, there are ℓ servers that can be queried. Each holds the same data, indexed in the same way. Clients query each one with a separate query, and can then recombine the

responses to extract the desired data. In simple PIR schemes, such as Chor's [7], the client only succeeds if each of the ℓ servers responds correctly. However, more robust schemes, such as those of Goldberg [16] and of Devet et al. [10], have since been developed that generalise this. In these schemes, only k of the ℓ servers need respond, and up to v of the servers may deviate from the protocol, and the client will still be successful in extracting their desired query.

We next sketch a design for a process to distribute a descriptor across ℓ servers so that multi-server PIR schemes are possible. Our system needs to allow for a single database to be distributed (identically) across multiple servers. When a client wishes to query for a descriptor, they must be able to figure out which servers can serve their query. We calculate the $\texttt{hsdir\_index}_i$ values as in the current Tor specification; however, previously these values pointed to the start of a sequence of `hsdir_spread_store` nodes, any of which could be queried for the desired descriptor. To support ℓ-server PIR, these will instead point to the start of the sequence of ℓ servers that will be used for the protocol.

When a circuit is constructed in Tor, basic precautions are taken to try and ensure that the routers in the path actually represent distinct, non-colluding entities. Specifically, the Tor Path Specification [34] outlines several constraints for building a path, which include the following:

- If two routers list each other in the "family" entry of their descriptors, they are in the same family and should not be in the same path.
- Two routers in the same /16 subnet should not be in the same path.
- Non-running and non-valid routers should not be in a path.

We can employ the same principles for path selection for the purpose of choosing the ℓ servers for multi-server PIR. Of course, any malicious adversary can simply ensure that routers they control do not list each other and are not in the same /16 subnet. These restrictions do not stop such adversaries, but take precautions to prevent incidental collusion by routers. We can take the exact same approach for choosing the HSDirs that will process a multi-server PIR query.

When a client wants to fetch a descriptor using multi-server PIR, it chooses a random index i from $\{1, \ldots, \texttt{hsdir\_n\_replicas}\}$ and computes $\texttt{hsdir\_index}_i$ for the hidden service. It then locates the ℓ next valid HSDirs who are all in different families and /16 subnets whose `dir_index` values come after $\texttt{hsdir\_index}_i$. The client can then engage in the multi-server PIR protocol with these ℓ servers. If this protocol fails for any reason, such as too many of the servers being unavailable, the client simply selects a new i and tries again.[3]

HSDirs keep their collection of descriptors separated into logical databases, according to their position in the sequence of ℓ servers in the hash ring; that is, the 'one' database held by an HSDir is identical to the 'two' database held by

[3] Currently, the number of times the descriptor is replicated (and thus the number of places it can be accessed from) is `hsdir_n_replicas` times `hsdir_spread_store`, which is currently 8. To ensure that there are the same number of logical databases where a descriptor can be accessed from, `hsdir_n_replicas` would have to be increased.

the next server, and so on. When a client makes their PIR query to each server, they must also indicate which logical database they are querying from, so that each database they query from is the same.

Note that for PIR schemes, it is crucial that all of the databases distributed across the ℓ servers are identical. Ensuring the consistency of replicated data across servers is of course a fundamental problem in the study of distributed systems. For this reason, specific solutions to the problem are largely orthogonal to this work. However, in Appendix C.1 we outline some of the general approaches that can be used in Tor in order to address this issue.

Perfect Hashing. In most PIR schemes, clients look up a particular *index* in a database without revealing that index to the server [7, 22]. Chor et al. [6] propose a number of ways to use a PIR scheme such that clients can look up records by a *string* instead of a record index. One of their techniques uses a construction called *perfect hashing*. Given a set of D keys (which are arbitrary strings), a perfect hash function (PHF) maps these D keys *injectively* into integers in the range $[0, \ldots, r-1]$, where $r = \mathsf{c} \cdot D$ and c is a small constant, typically in the range of 1 to 2. In our application, we choose to maintain a small c so as to maintain a smaller-sized PIR database. Ideally, we would like $\mathsf{c} = 1$, which results in a variant of perfect hash functions known as *minimal perfect hash functions* (MPHF). The information needed to evaluate an MPHF requires slightly more bits per key (D) to describe than a general PHF, but in our context this results in us being able to maintain a smaller PIR database size, which would intuitively result in overall gain. Therefore an MPHF seems like the ideal solution to our indexing challenge.

We discuss the details of the MPHF we use, provide benchmarks for it, as well as discuss why this does not entail any concerning leakages in Appendix D. Later in Section 5.1 we give optimizations to resolve indexing in hardware-assisted PIR schemes *without the use of MPHFs*, thus eliminating these leakages entirely in the hardware-assisted PIR case.

4 Privacy Analysis for PIR schemes

To evaluate candidate PIR schemes, we must discuss what underlying assumptions give a scheme its privacy properties, and how these assumptions hold up in Tor. The Tor context in no way affects schemes that make only computational assumptions, This means that single-server computational PIR schemes can be trusted to the extent that the underlying cryptographic assumptions are trusted. For XPIR and Seal-PIR, this corresponds to the Learning With Errors assumption, widely believed to be secure by cryptographers. The problem has received increased scrutiny and cryptanalysis due to post-quantum cryptography standardization efforts by NIST [26] and other standardizing bodies.

For hardware-based schemes, privacy guarantees depend on the security of trusted enclaves. In our implementation we leverage Intel SGX as the secure hardware module for our hardware-aided PIR scheme. Trusting the hardware in this case boils down to being able to verify that an HSDir that claims to support

hardware-aided PIR does in fact run on a processor with such hardware prowess. In Intel SGX, this is done via *remote attestation* [18], towards which Intel issues certificates that validate the claims made by its processors with SGX support. In the event of Intel "going rogue", they can at most misissue future certificates. This would not affect the security of PIR lookups that happened before misissuance. Additionally, we implicitly trust these modules to deliver the confidentiality and integrity guarantees they claim. However, recently researchers have demonstrated side-channel vulnerabilities of SGX that attack its confidentiality. These works have also demonstrated defences which are actively being incorporated by Intel, and this is a natural part of a new hardware component's lifecycle. In both of these classes of trust violations, our design still has *forward secrecy* in that the privacy of past queries cannot be compromised by future violations. Furthermore, even in such a worst-case event, the security of our scheme would simply reduce to the current status quo. We also note that while we used SGX to prototype our work, the underlying techniques can be adapted onto any of the other existing processors with secure hardware capabilities such as ARM TrustZone [3], AMD SEV [20], or their open-source sibling Keystone [21].

The privacy guarantees provided by multi-server ITPIR schemes are based on *non-collusion* assumptions made about the servers involved in servicing the queries. To guarantee the privacy of a query made by a client, we must assume that the ℓ servers involved in the query do not collude to break the privacy of this query. For non-robust schemes like Chor et al.'s, this assumption holds as long as at least one server does not collude to break a client's privacy. For robust schemes like Goldberg's [16], this is generalised so that as long as no more than t servers collude, privacy is still guaranteed. To analyze this assumption, we imagine an adversary who controls a Tor HSDirs, and then ask various questions about the probability they are able to break the non-collusion assumption. Remember that the position of a HSDir in the hash ring is determined by random values that the nodes have no control over, so that an adversary cannot adaptively position themselves in a hash ring to compromise security.

In addition to analyzing the probability an adversary can compromise privacy, we need to analyze the robustness of these schemes and the probability that the adversary can disrupt the availability of descriptors by behaving in a malicious way. In this section we will present our analysis on privacy compromise of queries, and in Appendix C we provide an in-depth analysis of availability compromise in the multi-server ITPIR model. With a multi-server scheme, the privacy can be compromised if the adversary is able to control at least $t+1$ out of the ℓ servers involved in the PIR query (with the value of t depending on the particular scheme). In order to evaluate how much more difficult this makes the adversary's task, we assume the adversary controls some number a of the n Tor HSDirs overall. We then ask the probability that when the hash ring is constructed, there is a consecutive sequence of ℓ servers in the hash ring (an "ℓ-block") where the adversary controls at least $t+1$ of them.

We can estimate how often this may happen using a combination of experimental and enumerative techniques. An exact enumeration is a somewhat

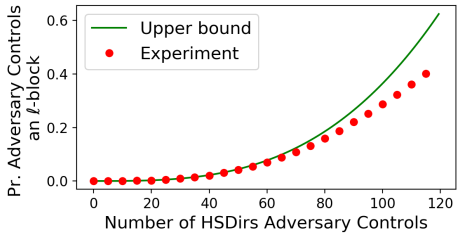


Fig. 1: Comparing our provided upper bound with experimental results for the probability an adversary is able to control *any* ℓ-block and violate query privacy in an ITPIR setting. Here $n = 4000$, $\ell = 5$, and $t = 2$. Experiments were performed by simulating 100,000 hash rings independently. Again recall that in the current Tor setup without PIR, this query privacy is *always* violated.

challenging combinatorial problem, but we can establish an upper bound. We provide an upper bound for the number of configurations of a hash ring with n nodes, of which a subset of size a are controlled by an adversary, such that at least one set of consecutive ℓ nodes contains at least $t + 1$ adversarial nodes:

$$U(n, a, \ell, t) := a \cdot \binom{a-1}{t} \cdot \binom{\ell-1}{t} \cdot (t)! \cdot (n - t - 1)!$$

This equation does not perfectly enumerate the number of hash rings where the adversary controls an ℓ-block. It overcounts this number of hash rings, because hash rings where the adversary controls multiple blocks are counted once per block. However, since it strictly overcounts, the equation can be used as an upper bound. We leave a tighter bound as future work.

Lemma 1. *The number of hash rings of size n in which an adversary controlling a nodes controls at least $t+1$ of a sequence of ℓ consecutive nodes is upper bounded by $U(n, a, \ell, t)$.*

We present the proof of this lemma in Appendix E. We can upper bound the probability that an adversary can compromise at least one database by dividing $U(n, a, \ell, t)$ by $(n - 1)!$, the total number of hash rings on n nodes. To see how this upper bound compares to the actual probability, we perform a series of experiments, varying the number of adversaries in the hash ring and observing the frequency with which the adversaries control an ℓ-block. With $n = 4000$, $\ell = 5$, and $t = 2$, we varied the number of adversaries from 0 to 120. Our results are shown in Figure 1. Notably, the lower a is, the better our upper bound performs compared to the true probability. This is not surprising, as our upper bound overcounts hash rings where the adversary controls multiple ℓ-blocks, which occurs more frequently when a is higher.

5 Benchmarking and Results

In this section, we discuss evaluations of selected PIR schemes we proposed earlier in Section 3. All our microbenchmarks are run on a single server-grade Intel Xeon E3-1270, with four physical cores, 64 GB of DDR4 RAM, and support for Intel SGX. Our server machine runs Ubuntu 16.04, and all our experimental results for the systems we measure are for a single core without any parallelism. For the end-to-end integration experiments we reuse the same server as the HSDir node, and use a 1.8 GHz i7-8565U laptop as the client. The source code to reproduce our experiments is available on our website.[4]

In order to get a more complete picture of how many lookup requests an HSDir handles at a time, as well as the sizes those descriptors, we monitored the activity on an HSDir for around eight months.[5] In terms of sizes, about 81% of the v3 descriptors we observed were <16 KiB, 7% were between 16 and 32 KiB, and 12% between 32 and 48 KiB. Hence for all of the PIR schemes we evaluate, we are concerned with large record sizes (approximately 16 KiB) since hidden service descriptors are about that size.[6] Larger v3 descriptors correspond to onion services that have a large encrypted list of authorized clients. To avoid leaking whether, and how many, authorized clients there are, hidden services may add fake lines to the descriptor to pad the length [35]. For microbenchmarks on multi-server PIR, we refer to Devet et al. [10], which provides a comprehensive picture on implementation details of many different configurations of multi-server PIR. Multi-server PIR schemes are fast, and we do not foresee the performance of these schemes being a bottleneck.

5.1 Hardware-assisted PIR Benchmarks

For hardware-assisted PIR schemes, we leverage ZeroTrace and benchmark four different variants of PIR flavours using it. Specifically, two variants of linear scan (one where the data is stored in the Processor Reserved Memory (PRM) pages, and the other where it is stored outside the PRM), Path ORAM, and Circuit ORAM. The linear scan variants do not face the indexing challenge, unlike the other PIR schemes. However, Circuit ORAM and Path ORAM do have the indexing problem to address. Instead of using an MPHF however, notice that in this context, indexing for the ORAM scheme can be achieved more simply by performing a linear scan over an array that maps blinded public keys of the hidden service descriptors to indices in the ORAM scheme. The overheads induced by this linear scan is minimal since each record in this array is a 32-byte key and an 8-byte index, and is significantly faster than scanning the entire

[4] https://crysp.uwaterloo.ca/software/piros/

[5] For privacy reasons, we only gathered bucketized numbers and sizes of descriptors, and never the actual descriptors themselves. See Appendix F for more information on how we followed Tor research safety guidelines.

[6] The smallest v3 descriptors are 14200 bytes; descriptors of this size are the overwhelming majority of descriptors in the < 16 KiB pool.

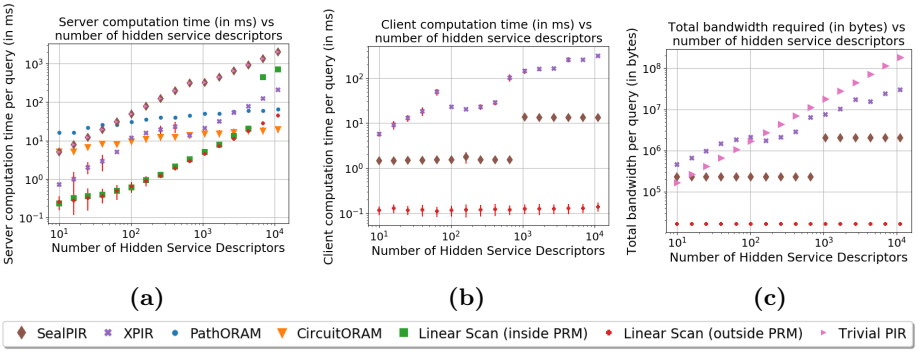


Fig. 2: Detailed microbenchmarks evaluating server computation time, client computation time, and total bandwidth overheads as a function of the number of descriptors held by the server. Trivial PIR requires no computation and is hence omitted from (a). For (b) and (c) we only display the line corresponding to ZeroTrace's Linear Scan with data stored outside PRM as representative of all the ZeroTrace variants since the client computation and bandwidth overheads of all the ZeroTrace variants are identical.

collection of hidden service descriptors. Our microbenchmarks for both ORAMs are hence inclusive of this online cost of index resolution via linear scanning.

We note that the PIR parameters for such a hardware-assisted ORAM scheme is simply a public key, under which the queries are encrypted by a client, such that only the SGX enclave can decrypt it. Hence these constructions also have the additional benefit that the tiny parameter size means they can forego the extra round trip required to fetch the PIR parameters, either by the optimization in Appendix B, or by simply including the public key in the Tor consensus directly. We observe later in Section 5.3 that doing so results in almost no perceivable overheads in the end-to-end latencies experienced by a user loading an onion service.

Our microbenchmarks in Figure 2a show that among the two linear scan variants, storing the data outside of the PRM scales better. Although counterintuitive, this arises from the fact that on Intel SGX, the PRM is limited to about 90 MB and thus when the data to be stored crosses this threshold, it leads to significant overheads induced by page faults. From our rough estimate in Section 2.1, the total number of v3 onion services that an HSDir server holds today is close to 1500. Hence for the remainder of this section we compare the schemes at the datapoint of 1702.[7] However, we note that both the linear scan variants of PIR in fact provide the best server computation time of 7.04±0.04 ms for 1702 descriptors. In comparison, the ORAM schemes are slightly more computationally expensive as seen in the figure with server computation times of 33.2±0.4 ms and 14.2±0.5 ms for Path ORAM and Circuit ORAM respectively at the same

[7] For the microbenchmarks in Figure 2 we chose data points evenly across the exponentially increasing x-axis, resulting in the odd data point of 1702 as closest to (but exceeding) 1500.

number of descriptors; however, as the number of descriptors increases, they soon outperform the linear scan variants. In terms of the bandwidth overheads induced, all four of these schemes are optimal since they leverage secure hardware at the server side, thus allowing the queries and responses to simply be AES encryptions of the blinded public key and hidden service descriptor (padded to 16 KiB) respectively.

5.2 CPIR Microbenchmarks

We show a detailed evaluation of XPIR and SealPIR in Figure 2 covering both computational and bandwidth overheads. In our experiments, we force XPIR to use LWE with 80-bit security, while tuning SealPIR's parameters to do the same. We allow XPIR's optimizer module to select the parameters d (recursion levels) and α (aggregation factor), for the best overall time for performing a PIR request. SealPIR's implementation is currently limited to small data record sizes; specifically, the implementation expects that a single data record will fit into a single plaintext polynomial, which limits an individual data record size to $1.5 \times \mathsf{N}$, where N is the degree of the underlying polynomial. (The $1.5 \times \mathsf{N}$ arises from the fact that with plaintext modulus $\mathsf{t} = 12$ bits, a completely filled degree-N plaintext polynomial can store exactly 1.5 bytes per coefficient.)

For large data records, one would store the data over multiple plaintext polynomials. Hence in our evaluations, we extrapolate SealPIR results by assuming that the costs of query processing and reply extraction will increase by a factor of K, where K is the number of plaintext polynomials required to store a single hidden service descriptor. To this end, we run our SealPIR experiments with a record size of 3000 bytes. (For $\mathsf{N} = 2048$, 3072 B is the maximum data size that a plaintext polynomial can store.) The query processing time (excluding the time for expansion) and reply extraction are then multiplied by $\mathsf{K} = 6$ to meet our required 16 KiB descriptor size. Figures 2a and 2b highlight the server computation time and client computation time induced by these schemes respectively. SealPIR and XPIR scale computationally poorly at the server side as well as the client side due to the underlying computation overheads of the FHE schemes (FV [15] and BV [5] respectively) that they use. The XPIR and SealPIR points in our graphs show irregularities since we allow the optimizer to select optimal parameters for each problem size. For all of our experiments we note that the XPIR optimizer chose to not use recursion, but instead heavily aggregate data blocks using high values of α (in the range of 8 to 60). In order to choose the optimal recursion point for SealPIR for a given problem size, we evaluate SealPIR with both choices of d and present the one corresponding to minimum total time in our graphs. The break in SealPIR points in these graphs correspond to switching the number of recursion levels (d) from 1 to 2.

At 1702 records of 16 KiB, XPIR induces a server computation overhead of 31±3 ms, and SealPIR 451±1 ms. These overheads induced by these CPIR schemes are barely practical today and higher than their ZeroTrace counterparts by about an order of magnitude. However the client-side overheads at the same datapoint (158±12 ms and 13.32±0.03 ms for XPIR and SealPIR respectively)

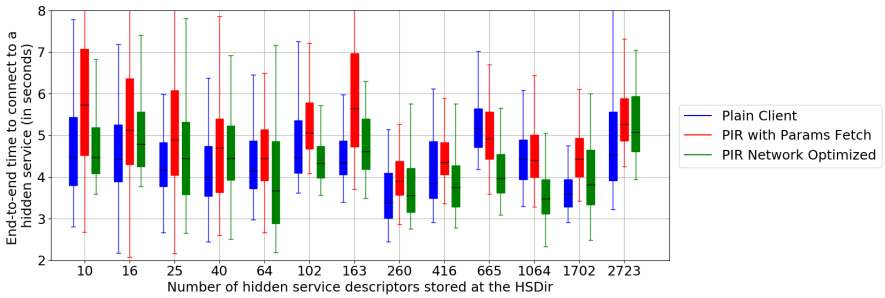


Fig. 3: Comparison of end-to-end latencies for connecting to an onion service with a plain Tor client vs. a Tor client with PIR support (with and without our optional optimization that saves the network round trip for fetching the PIR parameters). Our optimization saves approximately one second as we see from the difference in medians for the two PIR modes across the data points we collected. Moreover, end-to-end latencies for a user are barely impacted by incorporating PIR as the overheads of PIR are hidden by the noise of Tor network costs.

are more concerning as it may be much higher for lightweight clients like mobile users that would have weaker CPUs.

Queries in XPIR are encryptions of a bit vector of length corresponding to the number of records stored at the PIR server, with the bit at the index being queried being 1 (and 0s elsewhere). SealPIR introduces the notion of a compressed query, where the query is just an encryption of the queried index itself. In terms of reply extraction time, SealPIR and XPIR are quite close, and the difference in total client computation time arises from the fact that client query generation time is much smaller for SealPIR than XPIR, due to this query compression technique, but in total ZeroTrace is about two orders of magnitude better in terms of client computational overheads as seen in Figure 2b.

However this query compression technique has its own costs; first, it induces additional server-side computation for expanding this compressed query. Second, it limits the size of data that can be stored in a single underlying FV plaintext. Specifically, SealPIR has to force a plaintext modulus of $\mathtt{t} = 12$ bits out of its coefficient modulus of $\mathtt{q} = 60$ bits (as detailed by Angel et al. [2, §6] and seen in their implementation), so that after expansion and query processing the underlying plaintext is still decryptable with very high probability. The impact of this is not obvious from SealPIR's original evaluations, as they limit themselves to a small record size of 288 bytes, which fit within a single FV plaintext polynomial even with such low values of $\mathtt{t}$. Ultimately this technique makes SealPIR perform well in the context of large numbers of small records; however, this is the opposite of our context, where each HSDir has only a relatively small number of descriptors but of fairly large size, and this is reflected in our benchmarking.

Finally in Figure 2c, we see the total bandwidth overhead imposed by these schemes. Here we also include trivial PIR (clients download *all* descriptors whenever they make a query) as a baseline to compare the proposed PIR schemes against. At the datapoint of 1702 hidden service descriptors we see that XPIR request and response sizes are around 7.5 MB and 2.5 MB respectively.

While SealPIR is more bandwidth-viable than trivial PIR, it still requires about two orders of magnitude more bandwidth than any of the ZeroTrace counterparts. SealPIR alleviates the request size overhead using the aforementioned query compression technique. Thus both query generation time and size is significantly smaller than that of XPIR, at the same 1702 hidden service descriptors mark the request and response sizes are around 64 KiB and 1.9 MiB[8] respectively.

Concurrency and Computational Requirements. We note that both the CPIR schemes are parallelizable, and so are the linear scan variants of ZeroTrace, but not the ORAM counterparts. Hence multiple queries can be handled concurrently for these schemes. Even for the sequential ORAM schemes, multiple cores on the machine can be used to run several instances of the ORAM enclaves allowing it to serve concurrent queries.

We also collected the number of v2 and v3 lookups that our HSDir received during the months of April to June 2020. During this period, typically the HSDir served approximately 1500 queries in an hour. From the server computation microbenchmarks above and in the event that just a single core is used by the server to serve PIR, this would take less than 10.6 s in an hour to serve all these queries using the linear scan variant of ZeroTrace, while the XPIR and SealPIR require close to 46 s and 11.3 mins respectively to serve 1500 requests. However, some days we note spikes up to a maximum of 766,000 requests in an hour,[9] at which point the load cannot be supported by a single core alone; however, as we mention in Section 3, PIR operations should be handled asynchronously in a separate thread anyway. Using two cores on these HSDir machines asynchronously, even these peak loads can still be handled smoothly for the ZeroTrace variants. However, XPIR and SealPIR would require more than 7 and 96 cores respectively to handle such peak loads, making it prohibitive for deployment.

[8] Above ≈1000 descriptors, the SealPIR size jumps to about 1.9 MiB due to recursion, as seen in Figure 2c. Recursion in CPIR schemes increases the response size by a factor of f at each level, where f is the *ciphertext expansion factor*. Since the SealPIR compression technique reduces the effective plaintext modulus, it increases f from the expected ≈ 7 to 10. The expected $\mathsf{f} \approx 7$ arises from the fact that the underlying FV scheme can use a plaintext modulus $\mathsf{t} = 23$ for 80-bit security with a coefficient modulus q of 60 bits, but since SealPIR uses an effective plaintext modulus $\mathsf{t} = 12$ and $\mathsf{q} = 60$, and ciphertexts contain two polynomials, the total ciphertext expansion f is 10.

[9] These spikes were for v3 descriptors, and were presumably due to the HSDir holding an extremely *popular* descriptor, as we did not observe a corresponding spike in the *number* of v3 descriptors held.

5.3 Tor Integration Results

Finally, we also implemented and evaluated the impact on end-to-end latencies induced in Tor to connect to a hidden service when using PIR. For evaluating our proposal on the live Tor network, we ran a Tor relay that would serve as an HSDir, instrumented with our proposed modifications to support asynchronous PIR querying that we detail in Appendix A. Specifically, in addition to handling incoming HS descriptor stores as a normal Tor relay would, it also inserted the incoming HS descriptor into the PIR scheme's store. Similarly, we instrumented a Tor client to make PIR requests to this relay, and finally we created yet another Tor process that was modified to upload hidden service descriptors only to our HSDir with PIR support. In our experiments, we used this hidden service generator to generate several hidden services to a local web server, and then timed the curl requests for our client to perform a HS descriptor lookup and establish a connection with these hidden services. For privacy reasons, we only queried for the descriptors we ourselves uploaded.

With the above described setup, we evaluate three different clients; a standard Tor client, a Tor client that uses ZeroTrace's linear scan PIR (with the underlying data stored outside the PRM), and an optimized client that does not perform an additional round trip for the parameter fetch, assuming that the parameters were already available to the client as we describe in Section 5.1. We use the linear scan variant of ZeroTrace, since we know this to be the appropriate choice with the current scale of hidden services from our microbenchmarks in Figure 2a. We note that end-to-end latencies are subject to a lot of variance due to variability of several factors such as choice of relays for constructing circuits, and unpredictable network conditions encountered by different requests. Hence we present our findings in the form of a boxplot in Figure 3.

For each of the datapoints in Figure 3, we collected the timing reported for 100 curl requests to a hidden service that was not in the client's cache. The impact of our proposed optimization for compressing the additional round trip is immediately evident from this figure, as the medians for these two conditions seem to differ by almost an entire second across a majority of the datapoints we collected. Furthermore, we notice that deploying PIR in practice does not drastically impact the end-to-end latencies experienced by a user connecting to a hidden service, as the PIR overheads are completely hidden within the noise of overheads of using the Tor network.

6 Conclusion

HSDirs serve a unique purpose in the Tor network, acting as a DNS server for `.onion` addresses. For this reason it is important to make sure we can completely characterize the information HSDirs are privy to in their roles. We have shown that HSDirs have access to a relatively high amount of information in the Tor network. We find that the property of unlinkability, which is intended to guarantee that onion services cannot be tracked by HSDirs over time, is not provided

Table 1: Summarizing the results of all schemes that were considered

Scheme	Sec Guarantee	Required Changes	BW Overhead	Availability
Current	None	None	None	Unchanged
CPIR [1, 2]	LWE	Minimal	Large	Unchanged
ZeroTrace [30]	Hardware	Minimal	Minimal	Unchanged
ITPIR[7, 16]	Probabilistic	Major Changes	$\times\ell$	Increased with v

to all onion services due to the HSDir's ability to count the number of queries made for each service in an epoch.

To prevent this information leakage in the future, we investigate the integration of PIR into Tor. This integration is a complex problem due to the large design space and many PIR options, each with their own requirements, guarantees, and drawbacks. In this work we have thoroughly explored these options, explaining their strengths and weaknesses, and show how to integrate them into Tor. We conclude by discussing the options we have investigated and their suitability for our purpose. Our results are summarized in Table 1.

XPIR and SealPIR are attractive options because of their well-understood security assumptions (LWE) and the minimal changes needed to the structure of the hash ring. Clients would still only query a single HSDir in the hash ring, and the availability of the descriptors would be unaffected. However, the heavy computational and bandwidth burden incurred by XPIR is too high. SealPIR on the other hand alleviates the bandwidth overhead due its small query size, but worsens the computational overhead.

In contrast, Multi-server PIR schemes are very efficient, and the extra bandwidth used mainly comes from the fact that ℓ queries are made, instead of one. However, their security guarantees are probabilistic, and in each epoch, there is a possibility that enough adversarial HSDirs will be placed into an ℓ-block to compromise the privacy or availability of a query. Furthermore, multi-server PIR schemes require major changes to how descriptors are stored and queried.

Hardware-based PIR schemes are attractive in some senses, but challenging in others. Like XPIR and SealPIR, they require only minimal changes to the structure of the hash ring and process by which descriptors are queried. Availability is unchanged compared to the current state of the network. As well, these schemes perform very well, and add minimal bandwidth costs. The drawback with these schemes is that they depend on the security and availability of the hardware used. The security of trusted execution environments like Intel SGX is still being actively explored and improved.

Any use of PIR for retrieving descriptors improves privacy over the current state of Tor. Currently all queries (made by querying a blinded public key) are readable by an HSDir, allowing them to selectively deny queries and correlate incoming queries with the descriptors they hold to gather conclusions that erode the privacy of both clients and onion services. PIR can thus provide a significant improvement to the privacy of Tor onion services.

Acknowledgements

This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo. Edward Eaton was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) Alexander Graham Bell Canada Graduate Scholarship. Sajin Sasy was supported by an Ontario Graduate Scholarship and NSERC grant CRDPJ-534381. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

References

1. Aguilar Melchor, C., Barrier, J., Fousse, L., Killijian, M.: XPIR : Private Information Retrieval for Everyone. In: Proceedings on Privacy Enhancing Technologies (PoPETs). Sciendo (2016)
2. Angel, S., Chen, H., Laine, K., Setty, S.: PIR with Compressed Queries and Amortized Query Processing. In: Proceedings of the IEEE Symposium on Security and Privacy (S&P) (2018)
3. ARM: ARM Security Technology: Building a Secure System using TrustZone Technology (2015), https://developer.arm.com/documentation/PRD29-GENC-009492/c/TrustZone-Hardware-Architecture?lang=en
4. Belazzougui, D., Botelho, F.C., Dietzfelbinger, M.: Hash, displace, and compress. In: European Symposium on Algorithms. Springer (2009)
5. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In: Advances in Cryptology (CRYPTO). Springer (2011)
6. Chor, B., Gilboa, N., Naor, M.: Private Information Retrieval by Keywords. Cryptology ePrint Archive, Report 1998/003 (1998), https://eprint.iacr.org/1998/003
7. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private Information Retrieval. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS). IEEE (1995)
8. Danezis, G., Clayton, R.: Route fingerprinting in Anonymous Communications. In: Sixth IEEE International Conference on Peer-to-Peer Computing (P2P). IEEE (2006)
9. Danezis, G., Syverson, P.: Bridging and Fingerprinting: Epistemic Attacks on Route Selection. In: International Symposium on Privacy Enhancing Technologies Symposium. Springer (2008)
10. Devet, C., Goldberg, I., Heninger, N.: Optimally Robust Private Information Retrieval. In: Proceedings of the 21st USENIX Security Symposium (2012)
11. Dingledine, R.: Improving Tor's anonymity by changing guard parameters. https://blog.torproject.org/improving-tors-anonymity-changing-guard-parameters (2013), accessed March 2022
12. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The Second-Generation Onion Router. In: Proceedings of the 13th USENIX Security Symposium (2004)
13. Eaton, E., Sasy, S., Goldberg, I.: Improving the Privacy of Tor Onion Services. In: International Conference on Applied Cryptography and Network Security (ACNS). Springer (2022)
14. Elahi, T., Bauer, K., AlSabah, M., Dingledine, R., Goldberg, I.: Changing of the Guards: A Framework for Understanding and Improving Entry Guard Selection in Tor. In: Proceedings of the ACM Workshop on Privacy in the Electronic Society (WPES) (2012)

15. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144 (2012), https://eprint.iacr.org/2012/144
16. Goldberg, I.: Improving the Robustness of Private Information Retrieval. In: IEEE Symposium on Security and Privacy (S&P) (2007)
17. Hopper, N.: Proving Security of Tor's Hidden Service Identity Blinding Protocol. Tech. rep., The Tor Project (2013), https://www-users.cs.umn.edu/~hoppernj/basic-proof.pdf
18. Intel: Software Guard Extensions (Intel SGX) Data Center Attestation Primitives: ECDSA Quote Library API (2018), https://sgx101.gitbook.io/sgx101/sgx-bootstrap/attestation
19. Kadianakis, G.: Hidden Services need some love. https://blog.torproject.org/hidden-services-need-some-love (2013), accessed March 2022
20. Kaplan, D., Powell, J., Woller, T.: AMD Memory Encryption (2016), https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf
21. Karandikar, S., Devadas, S., Ou, A., Asanovic, K., Lebedev, I., Song, D., Lee, D.: Keystone Open-source Secure Hardware Enclave (2018), https://keystone-enclave.org/, accessed March 2022
22. Kushilevitz, E., Ostrovsky, R.: Replication is Not Needed: Single Database, Computationally-Private Information Retrieval. In: 38th Annual Symposium on Foundations of Computer Science (FOCS) (1997)
23. Loesing, K.: Distributed Storage of Tor Hidden Service Descriptors (2007)
24. Mathewson, N., Khuzhin, A., Provos, N.: libevent — an event notification library. https://libevent.org/ (2017), accessed January 2022
25. Mittal, P., Olumofin, F.G., Troncoso, C., Borisov, N., Goldberg, I.: PIR-Tor: Scalable Anonymous Communication Using Private Information Retrieval. In: 20th USENIX Security Symposium Proceedings (2011)
26. National Institute of Standards and Technology: Post-Quantum Cryptography. https://csrc.nist.gov/projects/post-quantum-cryptography (2019), accessed March 2022
27. Perry, M., Kadianakis, G.: Tor Padding Specification. https://github.com/torproject/torspec/blob/main/padding-spec.txt (2020)
28. Sanatinia, A., Noubir, G.: Honey Onions: A framework for characterizing and identifying misbehaving Tor HSDirs. In: IEEE Conference on Communications and Network Security (CNS) (2016)
29. Sasy, S., Goldberg, I.: ConsenSGX: Scaling Anonymous Communications Networks with Trusted Execution Environments. Proceedings on Privacy Enhancing Technologies (PoPETs) (2019)
30. Sasy, S., Gorbunov, S., Fletcher, C.W.: ZeroTrace : Oblivious Memory Primitives from Intel SGX. In: 25th Annual Network and Distributed System Security Symposium (NDSS) (2018)
31. The Tor Project, Inc.: Tor Metrics. https://metrics.torproject.org/ (2020), accessed March 2022
32. The Tor Project, Inc.: Tor directory protocol, version 3. https://gitweb.torproject.org/torspec.git/tree/dir-spec.txt (2021), accessed March 2022
33. The Tor Project, Inc.: Tor Guard Specification. https://gitweb.torproject.org/torspec.git/tree/guard-spec.txt (2021), accessed March 2022
34. The Tor Project, Inc.: Tor Path Specification. https://gitweb.torproject.org/torspec.git/tree/path-spec.txt (2021), accessed March 2022
35. The Tor Project, Inc.: Tor Rendezvous Specification - Version 3. https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt (2021), accessed March 2022

36. Van Den Hooff, J., Lazar, D., Zaharia, M., Zeldovich, N.: Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In: Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015). ACM (2015)

A Asynchronous PIR Computations

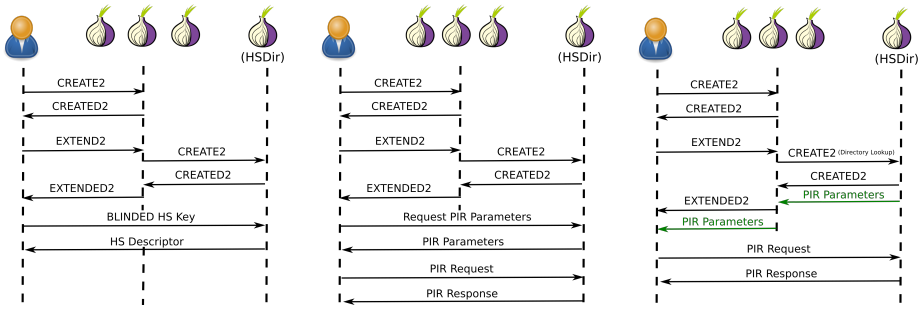
Integrating PIR into Tor's onion descriptor lookup mechanism requires some modifications to the internal architecture of the `tor` program itself. (We will use `tor` to denote the C program that Tor clients, relays, and onion services use.) The `tor` program does the vast majority of its work in a single thread, using an event-based paradigm based on libevent [24]. Currently, only the most expensive public-key cryptographic operations are shunted off to a separate worker thread. The computational costs of PIR (creating a query, processing a query, and processing the response) can be, depending on the PIR scheme used, far greater than those public-key operations, and so we should not block the Tor main thread, or even the cryptographic worker thread, while performing these computations. In addition, the PIR computations require libraries not currently used by `tor`, and we do not wish to add build dependencies to `tor` itself, nor risk introducing security vulnerabilities in `tor` if those libraries have flaws.

We therefore devise a new architecture for the onion descriptor lookup mechanism in `tor`. We create two additional binaries, `pirclient` and `pirserver`. These binaries do all the work of creating PIR queries and processing PIR responses (`pirclient`) and outputting the current PIR parameters, storing descriptors, and processing PIR queries to create responses (`pirserver`). When first needed, the `tor` process will launch one of these programs as a subprocess, and talk to it over a pipe using a very simple command-length-data communication protocol to minimize the attack surface. This pipe is then registered with libevent so that `tor` is notified when that pipe is readable or writeable.

We then alter the program flow for onion descriptor lookups, both to handle obtaining the PIR parameters, and to not block `tor` while the PIR computations are happening in the PIR subprocess. Operations that before were handled synchronously (for example, the HSDir looking up the descriptor corresponding to a given blinded public key and queueing it for delivery to the client) are now made asynchronous: in this example, the HSDir's `tor` queues the PIR lookup request for delivery to the `pirserver` subprocess instead. When a response from the `pirserver` subprocess is available, `tor` reads it, and queues the response for delivery to the client. Our modifications to Tor's onion service descriptor lookup architecture add a little over 1000 lines of code to `tor` (version 0.3.5.8). We note that while we advocate this architecture for the reasons above, other choices of implementation could of course be made.

B Round Compression

We first illustrate the network flows involved in the current onion service descriptor lookup protocol and describe modifications induced by our proposal.



(a) Current hidden service descriptor lookup protocol.

(b) Hidden service descriptor lookup protocol with PIR.

(c) Optimized hidden service descriptor lookup protocol with PIR.

Fig. 4: Network flow diagrams illustrating the current hidden service descriptor look up protocol and comparing it with a straightforward adoption of PIR for query privacy and our proposed optimized variant that saves a network round trip per request. For simplicity, we only display initiating CREATE flow, and the last EXTEND flow, omitting the intermediate EXTEND flows for setting up the entire circuit.

We then present an optimization that can reduce the additional round trip introduced by our proposal by piggybacking the retrieval of public information in the prerequisite circuit creation handshake flow.

Traffic flowing in the Tor network is sent in *cells*, which form the basic unit of communication. Each such cell contains a circuit ID, a command, a payload length, and a payload. Onion service lookup queries are performed by the client over a circuit to the HSDir it wishes to query. As illustrated by Figure 4a, once the client chooses an HSDir it wishes to query, in order to create this circuit, the client sends a cell with a CREATE2 command to its guard relay; this cell contains the first half of an authenticated handshake. The guard relay then completes the handshake and responds back with a CREATED2 cell, with the second half of the handshake. To extend this circuit, the client then sends an EXTEND2 cell to its guard relay, and the guard then performs a CREATE2/CREATED2 handshake with the middle node of the circuit, and finishes the EXTEND2 flow by responding with an EXTENDED2 cell to the client. This extend step is repeated until the circuit is extended all the way to connect to the HSDir.

Once the circuit to the HSDir is created, the client then performs the (non-private) query by sending the onion services's blinded public key for the current epoch to the HSDir through this circuit, and receives the corresponding descriptor from the HSDir. In order to make a query to an HSDir using PIR, the client needs to first learn the PIR parameters, implying the need for an additional round trip to provide privacy for the client's query (as opposed to privacy for the client's *identity*). Figure 4b shows the obvious straightforward implementation. However, interestingly the first round does not contain any information that depends on the client's query, i.e. this information could have been pub-

lished in the network consensus or as a part of the relay descriptor for HSDir relays, although that would result in significantly bloating up the network consensus or corresponding relay descriptor. Instead we propose a network-flow level optimization of piggybacking the first round with the EXTEND2 flow of the circuit construction, compressing this protocol back to a single round trip. The CREATE2 and EXTEND2 cells contain a two-byte HTYPE (Client Handshake Type) field that we propose to use to flag handshake flows that are explicitly for directory queries. The direct benefit is that it enables the HSDir to instantly identify if an incoming circuit establishment request is for a directory lookup, allowing them to piggyback a *public-information file* required for a client to complete its lookup; in our context, this file corresponds to the PIR parameters, which could include the description of the MPHF used (Figure 4c). We intentionally describe this as a generic public-information file, since this same trick can be used in the Tor network for relay descriptor lookups. Additionally, this allows for *incremental rollout*; HSDirs that do not support PIR will not respond with the public-information file, and the client can opt to use a different HSDir if private queries are important to them. This extension implies that the clients will now explicitly leak to the the nodes on its circuit that they are performing a directory query. This leakage can be obfuscated by using Tor padding state machines [27] for every circuit connection in Tor, at the expense of extra bandwidth. Additionally, we also note that if the PIR parameters themselves are small enough to fit in a single cell (the final CREATED2 cell that is returned from the HSDir), then we get this compression for free. We note that this is indeed the case for our hardware-assisted PIR scheme, as the parameter is simply a public key.

C Availability Compromise with Multi-Server PIR

As well as privacy, we need to consider how the use of PIR can affect the availability of onion services. We want to consider whether a malicious actor in the network can prevent an honest client from accessing a desired descriptor.

C.1 Availability Attacks from Descriptor Uploaders

A malicious uploader who wishes to compromise availability could accomplish this by uploading different descriptors to the ℓ HSDirs that form a logical database. It is important that the databases that exist on the ℓ separate servers be *identical*. Differences in the databases used in the multi-server PIR setting cause responses to be malformed. Different multi-server PIR schemes can tolerate different numbers (v) of malformed responses. For some schemes, like Chor's simple scheme [7], $v = 0$, and all responses must be perfect. We therefore need a way to ensure that the databases held by the HSDirs are identical. A straightforward approach to try to generalise how Tor's HSdir system currently works would be for the onion service to simply figure out the ℓ servers that are in

charge of their descriptor and send the descriptor to those servers, along with information about which of the ℓ servers they are.

The problem with this approach is that it means that *any* onion service can trivially shut the entire system down. By uploading different descriptors to the ℓ servers, all servers will essentially be 'misbehaving' and the response to any query made to that database, for *any* service, will be malformed. We need a new system that makes sure that the descriptors that the HSDirs hold are exactly the same. In Section 3 we discussed how HSDirs must also be consistent with how the descriptors they hold are indexed. For each collection of ℓ servers that manages a descriptor, that descriptor must have the same corresponding index. We have discussed how perfect hash functions [4] can be used to achieve this indexing. Therefore, HSDirs must have consistent perfect hash functions as well as a consistent set of descriptors.

Here, we focus on two high-level communication patterns that may be used to ensure that distinct servers have consistent databases. Both require HSDirs to communicate with each other to ensure that no one can upload different descriptors to a set of HSDirs. This means that all HSDirs must be capable of communicating with each other. These servers are therefore *collaborating* to *participate* in the protocol, but must not be *colluding* to *subvert* the protocol.

Onion Service Uploads Everywhere, Servers Check One possibility is to have the onion services upload the descriptors to the servers themselves (similar to the current structure). However, after upload, the HSDirs put the descriptors in a queue to be validated. At some predetermined interval (for example once an hour) the HSDirs validate all of the descriptors in their queue. This involves making sure that the descriptors perfectly match, and negotiating a perfect hash function to be used for the set of descriptors. If the method to generate the PHF can be made deterministic, then each HSDir can figure out the PHF for itself.

The advantage of this system is that it is conceptually similar to the current setup, except with a small amount of communication among the servers. Additionally, if one of the HSDirs is misbehaving (or offline), then it could potentially be identified by the remaining $\ell - 1$ servers.

Onion Service Uploads to First Server, Server Distributes An alternative is for the onion service to upload its descriptor to just one of the ℓ servers involved in the PIR scheme. That server is then charged with distributing the descriptor to the other $\ell - 1$ servers along with a new perfect hash function, if one is being used. The privileged server can do this as soon as it receives the descriptor, or once in every reasonable time period (say once an hour).

In this case, one server is put into a special privileged position above the rest. If we are using a non-robust multi-server PIR scheme, this does not affect the robustness of the system, because it is already the case that any server can trivially perform a denial of service attack on any query. However, it does allow for a malicious adversary who is chosen as this privileged HSDir to *selectively* refuse to distribute a certain descriptor, for example if they are aware what onion

service it is the descriptor for. If we are using a robust PIR scheme, then we lose many of the robustness guarantees by using this system, because whomever is chosen as the privileged HSDir can disrupt the system by misbehaving.

C.2 Availability Attacks from Malicious HSDirs

Multi-server PIR schemes are potentially more sensitive to an adversary who wishes to disrupt the system, as well as being sensitive to services being made unavailable by Tor relays unavailable for any other reason, including churn. For non-robust multi-server PIR schemes, every server involved needs to respond in order to recover the descriptor. In this case, a malicious relay (or more realistically, an offline server, or one that does not support PIR) can deny access to any descriptor they hold by not responding. It is important to remember that the adversary cannot *selectively* deny descriptors, because they do not have the ability to tell which descriptor is being queried, and deleting that descriptor from their database will similarly make *all* queries fail. We consider the probability that an adversary can cause a query to fail. For a robust scheme, where at most v Byzantine responses are tolerated, if a servers are controlled by an adversary then for a given query, the failure probability is

$$\left(\sum_{i=v+1}^{\ell} \binom{a}{i}\binom{n-a}{\ell-i}\right) \div \binom{n}{\ell},$$

calculated by the number of choices of the ℓ servers where at least $v+1$ are controlled by an adversary over all choices. For example, with $\ell = 4$, $n = 4000$, $a = 100$, and $v = 0$ there is roughly a 9.6% chance of a query being denied.

We can also consider the probability that an adversary is able to deny *any* query, rather than a specific one. For this case, our analysis that appears in Section 4 of an adversary's ability to compromise the privacy of a query can be used, but with v replacing t. Note that if the adversary wants to *selectively* deny access to the queries they receive, rather than denying all of them, they would also need to have compromised the privacy of the query by controlling t servers.

For many configurations of multi-server PIR schemes, the availability may be significantly decreased than single-server PIR schemes; due to i) the possibility of a malicious HSDir trying to negatively impact availability, or ii) simply churn in the HSDirs. To minimize the impact to availability of using a multi-server scheme, we may increase the number of times a descriptor is replicated. This is a consensus parameter of Tor, and can be increased. Indeed, increasing the number of replicas both helps with the vulnerability of the system to denial of service attacks or servers being unavailable (or not serving PIR), and also makes it so that the anonymity pools of the databases are larger. However, there is the negative effect of making the uploading process more cumbersome and putting a higher computational burden on the HSDirs.

D MPHF Details

In our work, we make use of the Compress-Hash-Displace (CHD) algorithm proposed by Belazzougui et al. [4], which is the most space-efficient MPHF construction in literature, and has an $\mathcal{O}(D)$ construction time for an MPHF of D keys and an $\mathcal{O}(1)$ evaluation time. We first evaluate the overheads induced to create and share an MPHF to overcome the indexing challenge. Consider an HSDir that has to store D onion service descriptors. On a 1.8 GHz i7-8565U laptop, the time taken to create an MPHF for 10,000 descriptors is less than 2.5 ms. The time scales almost perfectly linearly; as mentioned in Section 2.1, the number of onion service descriptors per HSDir is presently around 2000, which requires less than 0.50 ms. Additionally we note that this process of creating a perfect hash function only needs to be performed when new descriptors are added to the collection held by the HSDir. This can occur some constant number of times within each epoch, for example once an hour. Hence the time taken is negligible for an operation performed a small number of times in a one-day window.

As for the bandwidth overhead induced, the CHD algorithm provides very small descriptors (less than 5 KB) even for an MPHF for thousands of indices. For D items in the database, the description of the PHF is only slightly more than $D/2$ bytes. We note that this bandwidth would have to be transmitted for every onion service lookup in our model, but the size is clearly smaller than the size of a onion service descriptor itself[10], even if D increases to an order of magnitude larger than today, suggesting that even if there is an influx of onion services in the future without the number of HSDirs similarly increasing, MPHFs can continue to overcome the indexing challenge for using PIR for onion services without turning into a bottleneck itself.

We note that by the introduction of MPHFs for key-to-index resolutions, we introduce two new subtle leakages. First, the MPHF parameters supplied by an HSDir will inherently depend on the set of blinded public keys that the HSDir holds; i.e., the MPHF description introduces a leakage which in the worst case would enable a malicious client to reconstruct the entire set of blinded public keys held by the HSDir it queries. However, learning this set of blinded public keys is of no utility to the client, since to make any meaningful use of the descriptor for a particular blinded key, the client needs to know the underlying identity public key (or onion address) as we detailed in Section 1. Hence we argue that this leakage is inconsequential. Second, in any epoch a client can now discover how many hidden service descriptors a given HSDir currently serves. Although we cannot envision any privacy attacks from this leakage, this exact number can be hidden by honest HSDirs by either adding dummy hidden service descriptors to round up the number of hidden services they hold into discretized bins, or alternatively by sampling from a Laplacian distribution for differential privacy

[10] As noted in Section 5, the maximum size observed was less than 48 KiB.

guarantees and adding that many dummy descriptors, similar to how cover noise is generated in Vuvuzela [36].[11]

E Proof of Lemma 1

Lemma (1)**.** *The number of hash rings of n nodes in which an adversary controlling a nodes controls at least $t+1$ of a sequence of ℓ consecutive nodes is upper bounded by $U(n, a, \ell, t)$, where*

$$U(n, a, \ell, t) := a \cdot \binom{a-1}{t} \cdot \binom{\ell-1}{t} \cdot (t)! \cdot (n-t-1)!.$$

This upper bound for our hash rings is parametrized by four values:

- n, the total number of HSDirs
- a, the number of HSDirs that a passive adversary controls
- ℓ, the number of HSDirs involved in a PIR query
- t, the maximum number of HSDirs (out of ℓ) that can be controlled by the adversary without them being able to break the privacy properties (i.e., if $t+1$ are adversarial, they can).

For a quick summary of the proof, the terms in the expression for $U(n, a, \ell, t)$ represent the following choices that can be made:

- a: the number of choices for the first adversary-controlled node in the ℓ-block.
- $\binom{a-1}{t}$: the choice for the other t adversary-controlled nodes.
- $\binom{\ell-1}{t}$: which of the $\ell-1$ remaining spots in the ℓ-block are adversary-controlled.
- $(t)!$: the order for the adversary-controlled nodes.
- $(n-t-1)!$: the order for the other nodes in the hash ring.

Proof. We prove this lemma by demonstrating an *injective* function from the set of hash rings where this privacy compromise occurs to the set

$$S_{n,a,\ell,t} := [a] \times S_{a-1,t} \times S_{\ell-1,t} \times P_t \times P_{n-t-1},$$

where

- $[m] = \{1, \ldots, m\}$
- $S_{m,k}$ denotes the set of subsets of $[m]$ of size k
- P_m denotes the set of permutations of the elements of $[m]$.

The proof then follows from the observation that the size of this set is $U(n, a, \ell, t)$. Note that we will be demonstrating an injective function, but not a surjection. Some of the elements of $S_{n,a,\ell,t}$ will not be hit by this function, so $U(n, a, \ell, t)$ overcounts the number of hash rings where an adversary controls an ℓ-block.

[11] This Laplacian distribtution would have to be tuned to have a high mean, and low scale parameter to ensure non-negative values as detailed in Vuvuzela.

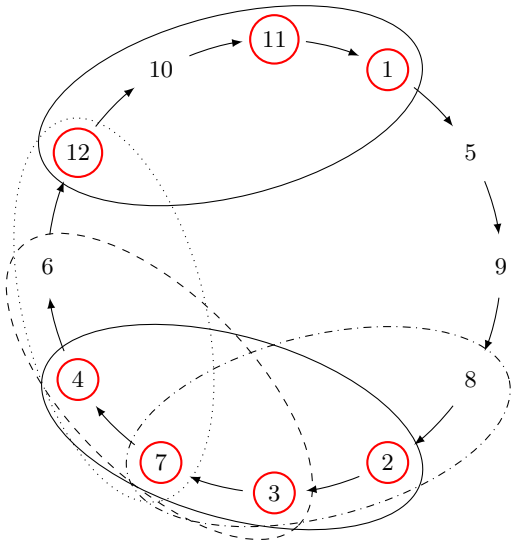


Fig. 5: An example of a random hash ring, with the parameters $n = 12$, $\ell = 4$, $t = 2$. Here the adversary controls the nodes $\{1, 2, 3, 4, 7, 11, 12\}$, so that $a = 7$ and the adversary controls 5 ℓ-blocks.

We are given n HSDir nodes, numbered 1 through n for simplicity, and a particular subset $\mathcal{A}$ of a of those nodes controlled by the adversary. We will count the number of permutations of the n nodes into a hash ring in which the adversary controls at least one ℓ-block.

To illustrate the proof, we will follow along with a simple example, depicted in Figure 5. Here the hash ring consists of just 12 nodes, 7 of which are controlled by the adversary. Here $\ell = 4$ and $t = 2$, and the particular arrangement causes the adversary to control a total of 5 ℓ-blocks.

For our injective function that maps from adversary-compromised hash rings to $S_{n,a,\ell,t}$ we will start by picking an element of $[a]$, as follows. We consider the ℓ-block that *starts* with an adversary-controlled node where this node has the *lowest* index amongst all adversarial nodes that start an ℓ-block. Continuing with our example, there are a total of 5 ℓ-blocks controlled by the adversary: $(12, 10, 11, 1)$, $(8, 2, 3, 7)$, $(2, 3, 7, 4)$, $(3, 7, 4, 6)$, and $(7, 4, 6, 12)$. Of these, the ℓ-block which starts with an adversarial node and this node has the lowest index is $(2, 3, 7, 4)$. We will denote the first node of this block $\mathcal{L}$, which is 2. To convert this element of $\mathcal{A}$ into an element of $[a]$, we denote the index of the node 2 in the (sorted) adversary set $\mathcal{A}$. So the element of $[a]$ we pick for the first component of $S_{n,a,\ell,t}$ is 2.

The next step is to choose an element of $S_{a-1,t}$, a subset of $[a-1]$ of size t. This ℓ-block is controlled by the adversary, so we know that there are at least

t other adversarial nodes in the block. Choose the t nodes with the lowest such label. This gives a subset of $\mathcal{A} \setminus \{\mathcal{L}\}$ of size t. We will map this so a subset of $[a-1]$ of size t by using the same indexing technique. In our example where the ℓ-block is $(2, 3, 7, 4)$, all of these are actually controlled by the adversary, and we choose the t $(= 2)$ with the lowest label, other than 2, which we have already considered. So, we pick 3 and 4. To get something in $S_{a-1,t}$, we note the the leftover elements of $\mathcal{A} \setminus \{\mathcal{L}\}$ are $\{1, 3, 4, 7, 11, 12\}$, so 3 and 4 are mapped to 2 and 3 respectively.

Our next step is to pick an element of $S_{\ell-1,t}$. This will indicate which positions in the ℓ-block the nodes selected in the last step take up. By indexing the remaining (after $\mathcal{L}$) positions in the ℓ-block as $1, \ldots, \ell-1$, we take the t positions that are taken up by the adversary nodes we have chosen. In our example, this is the first and third position, so we choose the subset $\{1, 3\}$.

Next, we pick an element of P_t. This denotes the order that the t elements we have chosen appear in the t slots we have noted. In our ℓ-block, we have 3 and 4 appear in that order, so the order might be written as $(1, 2)$, denoting that the node with the smaller label goes first. Our last task is to select an element of P_{n-t-1}. Note that we are left with the $n - t - 1$ nodes that we have not picked yet. Some of these may adversary controlled, and some may even appear within the ℓ-block. In fact, some may be both adversary controlled *and* within the ℓ-block, but were not 'selected' so far because their index was larger than the t adversary controlled nodes we chose.

All that is left to do is map these $n-t-1$ nodes down to the set $[n-t-1]$, using the same trick we did before mapping down to the index of the remaining elements, and specifying the order in which they appear. In our example, we are left with the nodes $\{1, 5, 6, 7, 8, 9, 10, 11, 12\}$. After $\mathcal{L} = 2$, and skipping over the nodes 3 and 4 which we have accounted for, these appear in the order $7, 6, 12, 10, 11, 1, 5, 9, 8$, which can be represented as an element of P_{n-t-1} by $(4, 3, 9, 7, 8, 1, 2, 6, 5)$. Our final element of $S_{n,a,\ell,t}$ is

$$\big(2,\ \{2,3\},\ \{1,3\},\ (1,2),\ (4,3,9,7,8,1,2,6,5)\big).$$

The proof of the lemma than follows from the injectivity of this function. This can be verified by noting that the process is reversible. Using the set $\mathcal{A}$ and the system parameters, we can always reconstruct the hash ring relative to the first node that we chose, $\mathcal{L}$, which is a unique choice.

However, note that we can only reverse this process for those elements of $S_{n,a,\ell,t}$ that were generated by applying this mapping in the forward direction. There may exist elements of $S_{n,a,\ell,t}$ that no hash ring will map to. This means that the size of the set actually overcounts the number of hash rings where an adversary controls an ℓ-block.

□

F Data Collection Mechanism

In this work we gathered data about two aspects of the onion services ecosystem from the live Tor network. We ensured our data gathering adhered to the safety guidelines listed by Tor Research Safety Board. We gather data on:

1. The number of descriptors an HSDir typically holds during an epoch, and the distribution of sizes of these descriptors
2. The number of descriptor lookup requests an HSDir serves over an epoch

We instrumented our Tor node to keep track of the number of descriptors it holds in an epoch for the descriptor size buckets we used in Section 5. At the end of every UTC day we bin them to a multiple of 50, and then log these binned counts for each of the buckets. For the second part, we instrument the same Tor node to keep track of the number of successful onion service descriptor lookups completed by it at an hourly granularity within an epoch, also binned to a multiple of 50.