Aviary: Distributed, Tamper-Proof, Per-User Warrant Canaries

Abhishek Bose-Kolanu

Aviary: Distributed, Tamper-Proof, Per-User Warrant Canaries

Abhishek Bose-Kolanu
Duke University
the.bobo@duke.edu

DISCLAIMER

This paper is a working draft. Cites, arguments and data may be incomplete or incorrect.

ABSTRACT

Governments routinely claim the power to subject individuals to secret investigation, forcing technology *Service Providers* to divulge *User* data without notification. Warrant canaries invert the notification problem by telling a *User* each time a *Service Provider* has *not* received a secret request for their data. Current canaries suffer from non-standardization, poor granularity, and brittleness in the face of attacks, leading the Electronic Frontier Foundation and Berkman Center to discontinue their Canary Watch service, which previously aggregated and monitored *Service Provider* warrant canaries, in May 2016.

Aviary is a distributed, tamper-proof, per-user warrant canary system intended to automate and replace obsolete canary practices. Aviary provides confidential, private, and secure warrant canaries with massively distributed auditing. This paper presents the Aviary system, analyzes it in the context of a threat model assuming a government-level adversary, and presents several mitigation strategies that inform the design of our distributed architecture.

To our knowledge, Aviary is the first scheme for global, distributed, confidential per-user warrant canaries, and among the first works to model a distributed system explicitly against the pervasive adversary outlined in the Internet Architecture Board's RFC 7624, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement."

1. INTRODUCTION

Research has quantified direct effects on *Users* from the threat of government incursion on privacy. Considered broadly, we might interpret such effects as part of a social "threat model" the present work attempts to address. In particular, it is troubling that mass surveillance effectively chills citizen research on precisely the issues given as necessitating more mass surveillance. A recent study found that views of Wikipedia articles related to terrorism dropped in a statistically significant fashion following the Snowden disclosures, both in the short term and over the long run, suggesting that internet users feared being flagged for secret government watch lists [25].

It is impossible for a free society to rationally debate policy tradeoffs when, as Glenn Greenwald writes, "1 in 6 writers [curb] their content out of fear of surveillance…Scholars in Europe [are] accused of being terrorist supporters by virtue of possessing research materials on extremist groups, [and] the British Library refuse[s] to house any material on the Taliban for fear of being prosecuted for material support of terrorism" [26] [27]. More recently, Google's automated account alerts triggered in a wave of suspected State-hacking attempts, many of which targeted journalists and professors in the wake of a contentious presidential election in the United States [17] [18] [19] [20] [21] [22] [23] [24].

However, political will among technology companies for protecting *User* privacy exists, and appears to be growing. The Electronic Frontier Foundation (EFF) and Berkman noted that the adoption of warrant canaries has increased by an order of magnitude, claiming that "[t]he last year has, without a doubt, been a banner year for the awareness of warrant canaries" [7]. We note that Facebook made its social network available over Tor, with over 1 million people making use of it [10] [11], Microsoft's President and General Counsel Brad Smith repeatedly challenges the legality of secret spying [12] [13] [14], Twitter wants to avoid the image of being cozy with secret police, as evinced by their efforts to reduce information leakage to the CIA [15], and Apple's CEO Tim Cook famously refused to hack into the company's own phones in the wake of the San Bernardino shootings [16].

Politically, from the perspective of technology *Service Providers*, will exists to protect *User* privacy and to adopt canaries. Notably, canaries are the only option available for companies wishing to notify *Users* of secret spy requests. Current law appears to support secret gag orders, forbidding companies from explicitly notifying subjects of secret government investigation. Warrant canaries invert this problem by having *Service Providers* instead notify *Users* whenever they are *not* a subject of a secret information request [1]. Canaries have additionally been at the forefront of legal challenges that have begun to question gag authority, placing some limits on gag duration and enforcement by allowing recipients to challenge gags and requiring the FBI to affirmatively prove the national security interest of each gag when challenged [89].

However, current canaries are reaching their strategic limits, and advances are needed. While previous warrant canary notices were high profile and numerous—including firms such as Reddit [2], Twitter [3], Lavabit [4], and the Google/Yahoo/Microsoft/LinkedIn/Facebook lawsuit resulting in the ability to publish ranges for secret requests received [5] [6]—current canaries face a strategic deployment problem. The EFF and Berkman Center shutdown their joint warrant canary aggregator in May of 2016, citing several challenges, including a lack of standard canary format, difficulty interpreting canaries, irregular tracking, and non-definitive information provided by canaries [7]. Absent an automated solution to these problems, the ability of canaries to scale is in doubt.

In this paper, we aim to create a new canary architecture, Aviary, that addresses these problems. Aviary provides a standardized, confidential and secure per-user distributed canary system. We wish to provide confidential (only the designated *User* knows if their canary is present), private (only the designated *User* can read their canary), secure (the *User* trusts the value of their canary) per-user warrant canaries. We further desire that the *User's* canary be anonymous, in the sense that attributing any given published canary to a given *User* is equally as likely as any other attribution.

Our traditional technical goals are to: (1) rule out passive pervasive surveillance as a successful strategy for determining if a *User* is aware of a secret spy request; (2) provide some safeguards against active network manipulation attacks targeting the Aviary system and *Users* of interest; and (3) harden the system against brute force active attacks, e.g. State-level DDoS. We have an additional goal we expand on momentarily, that of (4) limiting opportunities for transparent collusion between technology *Service Providers* and government agents. Our design architecture balances these privacy goals against the necessity of providing a performant solution for *Service Providers* and *Users* alike.

Here "performant" refers not only to standard engineering systems measures (computational cost, bandwidth expenditure, etc.) but also to designing a system that avoids inconveniencing *Users* as much as possible. Indeed, *Users* only interact with Aviary twice: once during key registration (Section 4.9), and in the event that their canary dies (Section 4.6). In these respects, Aviary advances the state of the art in massively distributed, government-resistant, and user-friendly privacy.

Our design avoids traffic fingerprinting to the extent possible while maintaining authentication to safeguard against active manipulation attacks, and uses smart addressing to frustrate common DDoS techniques, thereby addressing goals (1) through (3) above. Goal (4) bears some explanation. Per RFC 7258, "Pervasive Monitoring is an Attack," the general methodology to mitigate pervasive secret spying is to make it significantly more expensive and/or infeasible to perform without active collaboration, which risks detection and accompanying legal or business consequences [9]. Put plainly, secret spying is no longer secret when it becomes known. Under Aviary a government entity would not be able to request *User* data without Aviary alerting the *User* to the fact of the request. Accordingly, the government is left to attempt forcing the company to comply, in which case the constitutionality of its gag orders can be challenged in court (an outcome the government consistently avoids, see [89]). Otherwise, the government must find sympathetic or vulnerable entities *within* the *Service Provider* that they can successfully coerce into subverting Aviary.

This scenario has already played out, with disastrous consequences for the *Service Provider* in question. Recent revelations confirmed that executives at Yahoo actively circumvented their own security team to install a government backdoor into their email service. The backdoor was so poorly configured that: A) in a massive Fourth Amendment violation *all Yahoo emails* were intercepted and queried, not just investigative targets; B) it left the door wide open for external hackers to read *all incoming Yahoo email*; and C) in the words of a former Yahoo security engineer, when the security team found it, "they immediately assumed it had been installed by malicious hackers, rather than Yahoo's own mail team. (This says something about what the backdoor code may have looked like.)" [8].

The consequences were grave, with security engineers bleeding away to join rival firms, Yahoo's own Chief Security Officer Alex Stamos quitting to join in the same role at Facebook, and a massive public relations hit with two Congressmen voicing their displeasure, all at a particularly acute time as Yahoo attempted to close a sale to Verizon [90] [91].

Absent a system like Aviary, vulnerable individuals within a *Service Provider* face intense government pressure to facilitate mass privacy violations without any counterweight to protect them. In this regard, Aviary protects not only *Users*, but also *Service Providers*. It is worth noting that Aviary, like all warrant canaries, depends upon a *Service Provider* acting in the interests of disclosing secret spy requests to *Users*. Accordingly, nothing in Aviary's design prevents a rogue element within a *Service Provider* from disregarding it.

We argue that the existence of Aviary accomplishes three important goals in this regard. First, it offers a stable target for internal security teams to evaluate external delivery of *User* data against, raising the likelihood of discovery. Second, it changes the calculus for rogue elements and government agents alike, who must factor in the capacity for a *User* to be alerted to a secret spy request properly administered, or else hack the company. Finally, as live Aviary canaries constitute non-repudiable speech from *Service Provider* to *User*, in the event that a *Service Provider* lies about a secret search that becomes known, that lie is documented without the ability of the *Service Provider* to challenge it (see Section 5.5 "Cooperative Hosting and Archiving" for more).

We note that any implementation of a warrant canary mechanism remains at the forefront of legal precedent, with arguments on both sides regarding the legality of the practice [28]. Some legal scholars even suggest that widespread, granular warrant canaries may effectively 'flip' the legal debate, calling into question the legality of secret government gag orders within the United States [29]. While laws vary globally [30], current privacy advocates hold that the government cannot compel false speech, and as a result recommend the use of warrant canaries [1]. Noting that legal advice is well outside the scope of this paper, we nonetheless maintain that it behooves technology *Service Providers* to continue advocating strongly on behalf of *Users*, as it is providers who will be asked to implement policy and legal enforcement in cooperation with governments.

The remainder of this paper introduces our Threat Model and Design Goals in Section 3, provides a detailed explanation of our Canary Format and client software in Section 4, describes Distribution of Canaries in Section 5, and addresses some specific implementation details in Sections 6 and 7. An appendix addresses our choice of anonymizing network substrate.

2. RELATED WORK

Technical work on the deployment of warrant canaries appears almost non-existent. To our knowledge, Aviary is the first global, distributed, confidential and secure per-user warrant canary system in the literature.

However, there is a wealth of related work in the context of privacy enhancing technologies and hardening distributed systems, both against pervasive surveillance and provider interference. We draw on many of the mitigation strategies and attacks in the literature, citing related work throughout. Tor [31], i2p [32], and Tahoe-LAFS [33] provide principled ways of deploying secure distributed systems, such as minimizing metadata and data persistence and engineering with awareness of a pervasive network adversary, while work such as [34] shows how unthinking adoption of cryptographic pseudonyms alone does not guarantee privacy.

Specific work we draw inspiration from includes Google's Certificate Transparency efforts [35], a system using Merkle trees to monitor the Certificate Authority system for abuse, as well as work on measuring anonymity, e.g. thinking with anonymity sets and metrics [36]. Related work for hardening systems against the attack vectors defined in our threat model includes Octopus, an anonymizing DHT [37], Crosby-Wallach audit trees [38], and general DDoS mitigation tactics [39].

Our contributions include novel techniques specific to anonymizing warrant canary production and consumption (Section 4), hardening a distribution channel in a hybrid peer-to-peer and client/server approach (Section 5), and engineering a robust system that requires the absolute minimum of *User* interaction, all under the threat of a pervasive adversary with long-term passive and active capabilities.

3. SYSTEM OVERVIEW AND THREAT MODEL

Below we offer brief overviews characterizing Merkle trees, the actors in our distributed system, and the canary lifecycle before introducing our threat model.

3.1.1 Merkle Tree Properties

Merkle trees are hash trees that permit auditable, tamper-proof distribution of information over untrusted channels. Data is chunked into payloads associated with leaf nodes, whose hashes are computed. The term "leaf" is alternately used to refer to the leaf node or the data node, with context clarifying.

In the figure below, data nodes are designated *d0, d1, ..., d6*. Their corresponding leaf nodes are *a, b, ...f,* and *j*, whose values are simply a cryptographic hash of the corresponding data node. Successive intermediate nodes are built "bottom up" by applying the hash function to the concatenation of the two child nodes. For example, intermediate node *g* has a value of $H(H(a) \parallel H(b))$, where H is our cryptographic hash function and || denotes concatenation. The Merkle root, also known as the "tree head," is the resulting of this recursive hash procedure, and must be signed by the distributor. A recipient with a signed tree head and accompanying data nodes can reconstruct the hash tree to verify that data has not been changed or corrupted in transit. If the root cannot be reached, one or more leaf nodes has been corrupted.

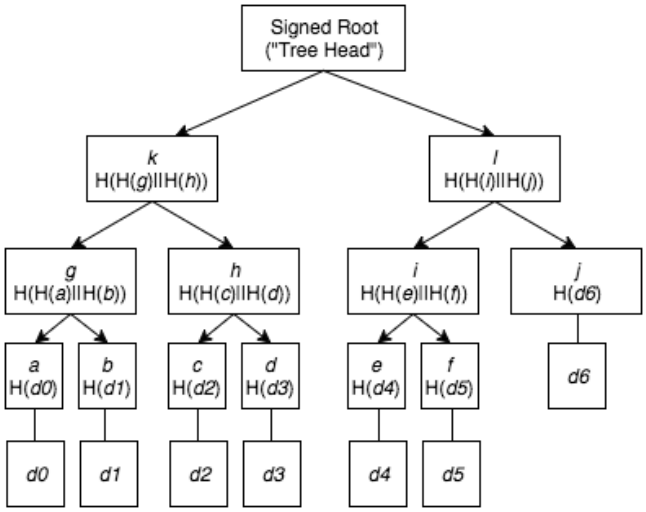

Figure 1. Sample binary Merkle tree with seven leaves. Adapted from Example 2.1.3 of [35].

Merkle trees also permit sparse verification, in which only a data node, signed tree head, and co-path are needed. Co-paths, or audit paths, are the hash values necessary to verify a node's presence in the tree. For example, in the tree above the co-path for *d6* is [*i*, *k*], while the co-path for *d3* is [*c, g, l*]. This property is the key insight behind our "binning" procedure detailed in Section 4.5. It also permits an interested auditor to begin auditing a *Service Provider's* published Merkle tree in an online fashion, without requiring a full download before verification can begin. Finally, Merkle trees also provide historically tamper-proof records, as node deletion or removal is not possible without altering the hash tree. Note that some subtleties regarding implementing Merkle trees are addressed in Section 7.1, "Choosing a Cryptographic Hash Function."

3.1.2 System Overview

The following sub-section identifies key players in the distributed systems architecture of Aviary. *Users* are end-users of a technology service, e.g. webmail, a social networking service, etc. *Service Providers* are technology companies providing services, from ISP's to social networks or backup and storage companies. *Service Providers* publish Merkle trees communicating canary information to *Users* once each publication period (e.g. once a month). Remarkably, Merkle trees permit anyone to audit them without endangering the privacy and confidentiality requirements set out above. Aviary trees are subject to repeated and automatic system-wide audits at the hands of the *Users*.

Tree Hosts, run by *Service Providers* and interested volunteers, are seed peers in an anonymized BitTorrent swarm acting as a distribution mechanism. A *Key Service* is a third-party service providing identity proofs for *User* ownership of public keys, which *Service Providers* use when constructing canaries. i2p is an anonymizing mixnet over which the Aviary architecture runs.

Wherever TLS certificates are mentioned, we follow RFC 7624 Section 5.2, "Attacker Costs," and recommend implementation of RFC 6962, Certificate Transparency, to strengthen the TLS certificate chain [40] [35].

Finally, we note that the system, device, and region diversity of large *Service Provider* userbases (i.e. O(1 billion *Users*)) proportionally increases the strength of auditing for their canary trees. While concerns over backdoored hardware, operating systems, and supply chains exceed the focus of this paper, it is worth noting that the massively distributed nature of the Merkle audits being performed lends higher credibility to the integrity of Aviary warrant canaries.

3.1.3 Canary Lifecycle

We present a brief overview of the canary construction, publication, distribution, and consumption lifecycle. Canaries are tied to specific *Users* and come in one of two states: live, or dead. Since canaries cannot constitute additional speech under current law, dead canaries are simply absent, or missing, canaries [47]. Thus, under Aviary, a dead canary is one that is not published. *Service Providers* also publish a second kind of canary, "dummy canaries," which are not tied to any *User* and are explored in Section 4.

We do not assume any special security literacy or action on the part of the *User*, and the vast majority of their interaction with Aviary is automated. Once the *User* registers a public/private keypair with an attestation service (a third-party *Key Service* that provides identity proofs for key ownership), their subsequent responsibilities are nil. A *Canary Locker* (Section 4.6) component in their web browser automatically handles updating a client salt with the *Service Provider* (a key component for randomization of canary placement in the *Service Provider's* Merkle tree) and pulling new canaries each publication period. These network transactions occur over *i2p*, an anonymizing mixnet with end-to-end encryption that severs IP addresses from routable destinations and frustrates passive fingerprinting.

Service Providers publish three Merkle trees each publication period: a canary tree, a key tree, and a tree of record. The canary tree contains *User* canaries, while the key tree is a rapid indexing tree that solves some scaling challenges as well as providing extra anonymization benefits, as the *Service Provider* does not persist a

mapping of canary to *User*, and thus minimizes risk of *User* de-anonymization (Sections 4.4, 4.7). The tree of record provides contiguity between each publication period's canary and key trees, providing non-repudiability across time (Section 5.4).

This tree material is distributed via an anonymized and authenticated BitTorrent swarm running over i2p (Section 5.2.1). *User* Canary Lockers coordinate participation in a multi-*Service Provider* DHT (distributed hash table), while volunteer or *Service Provider*-run *Tree Hosts* act as full seed peers on the swarm, bootstrapping tree distribution (Section 5). *Users'* Canary Lockers unpack the canaries they have downloaded to search for a valid canary belonging to their *User*, and in the process audit the Merkle tree to verify that tree material (canaries) have not been manipulated. In the event that a *User's* Canary Locker cannot find a valid canary for a given *Service Provider*, the browser displays a persistent banner warning alerting the *User* that their account for that *Service Provider* may be subject to secret investigation.

Many details concerning implementation, DDoS-hardening, and specific tricks involved in realizing anonymity and privacy gains are omitted, but we hope this bird's eye view helps orient the reader for the work ahead. We now proceed to introduce our threat model.

3.1.4 Threat Model and Design Considerations

We follow the language of RFC 7624, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement" [40]. We assume an adversary with the following three capabilities and interests. First, massive pervasive passive monitoring of the entire internet, including post-hoc analysis of recorded data. Second, active monitoring of some portion of the internet, including ad-hoc manipulation, deletion, or injection of messages, and/or impersonation, particularly in the case of targeted *Users*. For example, a State actor may wish to know if a target of a secret government investigation has successfully been alerted through the use of an Aviary canary. Third, State-level DDoS against core infrastructure items of the Aviary system. We assume that the communication endpoints, in other words *Users* and *Service Providers* (or, at a minimum, the internal security teams of *Service Providers*), do not collude with the adversary, but do permit the adversary to present malicious "user" nodes.

We assume an adversary without quantum computing capability, though we note throughout the text where this assumption may be worrisome and offer some mitigating strategies. Our approach consists of a "defense in depth" model, the combination of cryptographically sound primitives, and targeted usage of distributed systems architectures engineered for security and privacy.

We wish to provide confidential (only the designated *User* knows if their canary is present), private (only the designated *User* can read their canary), secure (the *User* trusts the value of their canary) per-user warrant canaries. We further desire that the *User's* canary be anonymous, where we define anonymity as the maximally entropic distribution over elements in a set (e.g. random uniform distribution). Hence, we seek to expand the *User's* anonymity set (the number of canaries her canary mingles with) wherever possible, while balancing this goal against the engineering tradeoffs involved.

We assume trust in the *Service Provider's* intentions, and weakly assume trust in the integrity of their TLS connection, though this trust is as minimal as possible (see the salt pin message in Section 4.7 "Communicating Bins to Users").

Passive attacks are by definition not detectable at either endpoint of the communication medium (assuming neither endpoint is a collaborator), but permit an attacker with sufficient resources to record communication for later analysis, e.g. when an investigation indicates a particular individual of interest, when a new analysis technique permits fingerprinting an exchange that was previously not easily detected, or when the advent of a quantum computer permits cracking TLS keys previously assumed safe against classical attacks.

Per RFC 7624 we assume the adversary has the capacity and capability to observe all packets sent in the internet and observe all data at rest in any intermediate devices between endpoints. We further assume the possibility of an active pervasive attacker in the case of *Users* or popular *Service Providers* subject to secret government investigation. We restrict our interest in this active scenario to the integrity of security guarantees only insofar as the warrant canary system is concerned.

While we note that such *Users* and *Service Providers* are likely to be subjects of advanced persistent threats, or so called "network investigative techniques" (NIT's) as what appear to be at work in a case of the FBI allegedly mass hacking users of TorMail without regard to the scope of their warrant (over 8,000 users in 120 countries on the basis of a single warrant [41], [42], [43] and [44]), we do not address such ancillary threats in this work, treating the security of internal systems for both *Users* and *Service Providers* as outside the scope of our analysis. In general, our method assumes an untrusted core and pushes security competency to the edge as much as possible, with analysis restricted to the case of a global, trustworthy, per-user, confidential, and private warrant canary system.

4. WARRANT CANARY FORMAT

4.1 Current Challenges

Current warrant canaries face three significant challenges. Their format is not standardized, leaving *Users* and *Service Providers* to reinvent the wheel each time a warrant canary is deployed. Some examples of current canaries include automated postings to Twitter accounts, color schemes for *Service Provider* logos, and statements in regular corporate reports, such as Annual or Transparency Reports. Since there is no standard canary, adoption and user literacy are both hampered.

Second, canaries tend to be extraordinarily coarse. For example, Reddit has over 10M monthly unique visitors in 2016, but its warrant canary provided no information on which accounts might be affected when it sprang earlier this year [45] [46]. Coarse canaries are not informative. Once the canary is triggered no *User* has a good idea if they were caught up in the dragnet or not, and the canary becomes unusable for future requests. This scenario also incentivizes a government agency burning a coarse canary on a dummy request and then sending legitimate requests after the *Service Provider* loses the ability to meaningfully signal the occurrence of subsequent snooping attempts.

Finally, canaries are both centralized – in the sense that they are generally hosted on a single page – and fragmented, in the sense that no *Service Provider* benefits from the existence of other *Service Providers'* canaries.

These design decisions minimize canaries' utility, as *Users* are left to interpret changes in a canary status on their own, without a reliable audit trail (was that logo always blue or did it change? Did it actually change or was that a momentary bug?), and *Service Providers* are unable to offer independent proof-of-record of a canary's status over time, nor are they able to easily insure a canary against a determined DDoS attack.

4.2 Aviary Canary Format

Warrant canaries in Aviary are formatted as leaf nodes in a *Service Provider's* canary tree. Aviary canaries must be easy for client software to interpret, private such that only the *User* can read their own canary, and confidential such that only a *User* can know if there exists a canary corresponding to their account. In addition, we want to satisfy these conditions within a tractable engineering design space, and in a cryptographically secure fashion. Finally, to comply with current law, "dead" or "tripped" canaries cannot constitute additional speech to notify their owners of a secret government request. Dead canaries must be silent.

We accomplish privacy through the use of public-key cryptography, as described in Sections 4.4 and 4.5. We now proceed to describe our standardized Aviary canary format.

An Aviary canary is 32 bytes long. The first half consists of 16 bytes for the first 16 characters of the *User's* username. If the username is shorter than 16 bytes, a pad of length (16-*h*) where *h* is the length in bytes of the username is appended. The second half consists of a 16-byte cryptographically secure nonce, unique (random) within a *Service Provider's* publication set. Canaries are encrypted with the *User's* public key.

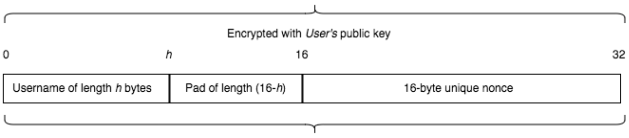

Figure 2. Aviary Canary Format

Constant length canaries are a requirement to avoid leaking information about which usernames do or do not have current canaries published. The nonce is a requirement to ensure that a given *User's* canary status cannot be tracked. Without a nonce an interested third party could retrieve the *User's* public key and encrypt their (username + pad) combinations until finding a match among the canary tree's leaf nodes.

Beyond de-anonymization of the canary, repeated publication of the same encrypted canary across publication cycles would also constitute a beacon, allowing an interested third party to know if a canary is still live, along with when a *Service Provider* has nullified a specific canary. Combining this information with the above de-anonymization attack, or with private information an interested third party may be privy to (e.g. timeline of a secret government request to *Service Provider*), could permit de-anonymization of the canary, and consequently knowledge of the *User's* knowledge or lack thereof of a secret spy request for their information. Thus, we require nonces that are random across publication cycles.

4.3 Dummy Canaries

What should a canary do when it "trips" for a given *User*? For our purposes it is sufficient to permit *Service Provider* to effectively notify *User* so long as *Service Provider* can do so without producing any additional speech that causes this effect.

Accordingly, when a given *User's* canary trips, the *Service Provider* simply stops publishing a canary for them. However, government desires to obfuscate the true number of secret spy requests made annually require *Service Providers* to avoid disclosing an exact count of NSL/FISA requests received, instead publishing ranges like "0-249" or "1000-1999" [47].

If a *Service Provider* simply refused to publish canaries for compromised *Users*, an interested observer could subtract the size of a *Service Provider's* Merkle tree from one publication cycle to the next to derive the count of compromised *Users*, assuming no growth or churn in the *Service Provider's* userbase, or equivalently, a good enough estimate of *Service Provider's* net userbase growth between publication cycles (see Section 6 for a full treatment).

Consequently, *Service Providers* must publish *dummy canaries* to replace any dead canaries of *Users*. A dummy canary has the same format as described in Section 4.2, with the exception that the canary is composed of two 16-byte nonces. In place of the *User's* public key, a random public key must be used to encrypt the canary. Note that the requirement for a random public key is necessary, as using the *User's* public key might constitute speech notifying a subject of a secret government investigation.

4.4 Client Verification of an Aviary Canary

The most client-optimal strategy, in terms of minimizing the amount of work a *User* must perform, would have the *Service Provider* send the *User* their canary node directly. This solution violates our security guarantee of auditability, as a *User's* canary no longer forms part of a public Merkle tree that leaves the control of the *Service Provider*. It also raises the attack profile for the *Service Provider*, as it would require the *Service Provider* to persistently associate wrapped canaries with *Users* instead of processing them as a one-time pass-through service (e.g. nonces never leave RAM, and are immediately wiped after processing). Additionally, transmitting the wrapped canary to client software would provide another vector for re-personalizing the canary, as a passive third-party listener on the network would see only one wrapped canary passing from *Service Provider* to a given *User*. In the event of a tripped canary, if the *Service Provider* sent a dummy canary directly to the *User* to avoid passive detection, this may violate law by constituting speech notifying a *User* of a secret request.

At the opposite extreme, a maximally agnostic strategy would involve the *User* downloading the entirety of the canary tree. With a *Service Provider* that serves 1 billion users, at 32 bytes per canary, this would correspond to at least 32GB of data, an unwieldy amount for many home internet data plans and end-user devices. We present our solution shortly, in Section 4.5, "Client Binning." For now, we describe how client software verifies a canary.

Algorithm 1: Client Canary Verification

```
Given a set S of candidate nodes.
canary_flag = False
while S is non-empty:
    select some node s from S
    msg = s.decrypt(UserPrivateKey)
    if msg[0-15] == first16CharsofUserName
        canary_flag = True
```

A client function receives a node. It decrypts the node using the *User's* private key. If the first 16 bytes correspond to the first 16 characters of its username, the canary is valid. If a *User* processes all nodes in a publication set and does not find a valid canary, the client software alerts the *User* that their account has been compromised. Note that we avoid early termination of the verification sub-routine upon discovery of a valid canary as a matter of good security hygiene.

In the extremely unlikely case of a key collision between a dummy canary and a *User's* public key, the client software successfully distinguishes the two by looking for its username in the first 16 bytes of the unwrapped canary.

4.5 Client Binning

To address the challenge of balancing client work with maintaining anonymity of the *User's* canary from the previous section, we propose a method we call *client binning*.

A *Service Provider* should "bin" each *User* into a bin by means of a fast, randomized hash assignment. A cryptographically secure hash is not required, as the nonces produced in Section 4.2 are already cryptographically random. Accordingly, we are free to make use of the fastest random hash available, and suggest xxHash or t1ha, each of which runs at near-RAM speeds [48] [49]. Upon drawing of the random nonce and construction of the *User's* canary, a *Service Provider* passes the encrypted bundle as an argument to the hash function for hashing into a randomly assigned bin.

Bins correspond to sub-trees of the *Service Provider's* Merkle tree. The leaf nodes of a sub-tree belonging to a given bin are composed of the canaries for each *User* in the bin, including dummy canaries. Thus, the interior node at the "root" of the bin is a Merkle root for the sub-tree whose leaves are the canaries of *Users* (or dummies) belonging to that bin.

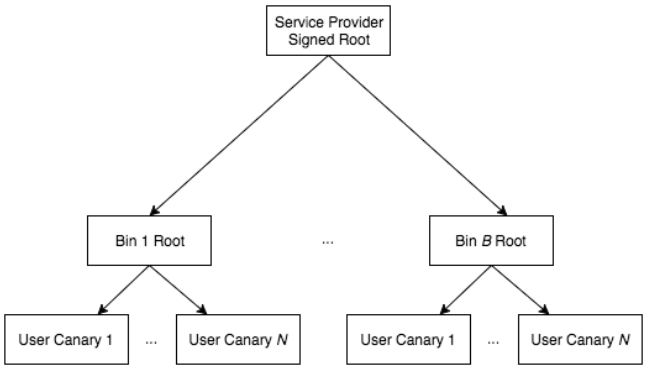

Figure 3. Bins for a single Service Provider, intermediate nodes omitted.

We note that maximum entropy within a publication set is achieved when the probability that any canary belongs to a given *User* is uniformly distributed among the *Users*. That is, $P(c, u) = \frac{1}{U}$, where $P(c, u)$ is the probability of a given canary c belonging to a given *User* u, and U is the total number of *Users* [36].

We can characterize the additional marginal anonymity benefit of expanding a *User's* bin explicitly: for each additional record added to the bin, the *User's* anonymity increases by a quantity directly proportional to the difference $\frac{1}{N} - \frac{1}{N-1}$, where N is the number of *Users* in the expanded bin.

As the *User* must download the additional canary, we can cost this gain in anonymity in terms of the additional 32 bytes the *User* must download (ignoring for the moment additional overhead produced by encrypting the canaries and computing non-leaf Merkle nodes). Each additional canary in a bin purchases diminishingly fewer gains in anonymity.

In the interests of balancing download size with anonymity, we declare a bin size of 100,000 to provide sufficient anonymity for a *User*. In this case, the total size of a bin downloaded by a *User* would be on the order of 3.2MB, or roughly the cost of one streamed song. We note that the probability a given canary in the bin belongs to a given *User* is at most $\frac{1}{100{,}000}$.

"At most" because some *Users* in the bin may have already been compromised, in which case some nodes $n \in N$ are actually dummy canaries. Additionally, in the event that a *Service Provider* is unable to fill a bin with 100,000 *User* canaries, dummy canaries may be used to fill the bin size, such that all *Users* receive the minimum anonymity measure through equivalent bin sizes.

We note that a *Service Provider* should randomize bin assignment at the start of each new publication generation, so that a given *User's* bin does not become a stable identifier across publication generations. We cover bin assignment in more detail in Section 4.7, "Communicating Bin Assignments to Users."

4.6 Client Canary Lockers

Where should the client software live? Our ideal solution is automated and device agnostic, working across both laptops and mobile, as many *Users*, particularly those in emerging markets, are mobile-first. The solution we propose is largely workable on mobile, though some subtleties (timing of data access, scheduling compute around battery life and network reliability) are omitted for brevity. We assume deployment on desktop/laptop for the remainder of this paper.

We propose that the major browsers implement a standardized Aviary Canary Locker. Periodically, the Canary Locker retrieves the appropriate canary sub-trees, or bins. The browser then handles querying the *User's* operating system's keystore for *User's* private key, unwraps the nodes, and checks for presence of a live canary. In the event that no canary is found for a *Service Provider* publication period, the browser can present a persistent banner warning at the top of the browsing window, advising the *User* that their account with the given *Service Provider* may be subject to secret government spying. All operations can be handled automatically, without need for *User* intervention.

The advantage of implementing through the browser is three-fold. First, browsers are well-tested pieces of software that invest heavily in sandboxing and other security measures. Second, most non-IoT devices a *User* has ship with a browser: laptops, phones, tablets, and even some e-readers. Finally, browsers are already engineered for cross-platform compatibility.

Service Providers must tag their published canaries with expiration dates, so that the Canary Locker can validate whether or not a given device has checked against the latest available generation (see Section 5.1 "Tree Publications"). This also permits a Canary Locker to notify a *User* for how long their canary has been tripped.

Finally, we note that Canary Lockers continue to download and process bins for a given *Service Provider* even after a canary trips. Accordingly, *Service Provider* must assign bins each publication period not just to *Users* with live canaries, but to all *Users*. This has the benefit of strengthening the auditing of the tree, and also prevents a third-party network snooper from divining that the *User* is aware their canary has been triggered, as network access patterns do not change post-trigger.

4.7 Communicating Bin Assignments to Users

Service Providers publish three bundles each time they publish a full set of canaries. The first bundle consists of a signed Merkle tree root for the canary tree, along with the leaf nodes for that tree. For a large *Service Provider* this bundle has O(1 billion) leaf nodes, for a total size of ~32GB. We include a rapid indexing bundle to avoid downloading the entire canary tree (and to avoid *Service Providers* persisting the map of canary to *User*, per Section 4.4). This second

bundle consists of a secondary Merkle tree we call the "key tree."[1] *Service Providers* publish the key tree's signed root and its leaf nodes. The third bundle, the tree of record, is explored in Section 5.1.

The key tree has as many leaf nodes as there are bins for the canary tree. This is given by dividing the total number of canaries published by the bin size, which in our working example evaluates to 1 billion/100,000 = 10,000. The payload for each leaf in the key tree is a list of 32-bit integer offsets. Half of each integer is available for enumerating bins. The length of the list is equivalent to the number of canaries in each bin, and each entry is encrypted with the public key of one of the *Users* (or dummies) in that bin.

Each offset acts as the bin identifier for the canary tree. For example, a 0 corresponds to the canary sub-tree composed of canary leaves in the range [0, 9,999]. A 1 corresponds to the next bin, composed of canary leaves [10,000, 19,999]. A 2 corresponds to the bin [20,000, 29,999] and in general an n corresponds to the bin [n**bin_size*, (n+1)**bin_size*-1].

The size of each leaf in the key tree for a *Service Provider* with 100,000 *Users* per bin is ~400KB, ignoring encryption overheads. The total size of the key tree in this scenario would be 4GB. While certainly less than the 32GB canary tree, this size is still unwieldy.

Fortunately, *Users* have a method for determining which key leaf contains their bin identifier, and may download only it instead. *User* and *Service Provider* both use a known random hashing algorithm (e.g. xxHash) to determine the mapping of key leaf to *User*. They both hash on the concatenation of *User's* public key and a client-supplied salt to derive the mapping. We explain our method for secure communication of the salt momentarily, focusing for now on how the salt is used.

The *User*-submitted salt acts as a bin identifier hash salt, such that `math.floor(xxHash(UPK || CLS) % B)` yields the index of the key tree leaf for that *User*, where || denotes concatenation, *UPK* is *User's* public key, *CLS* is the Canary Locker salt, and *B* is the total number of bins for that *Service Provider*. Unpacking the payload of the designated key tree leaf reveals the list of 32-bit integer offsets. The Canary Locker iterates through the list and decrypts each entry with the *User's* private key until the bin identifier is found. Valid bin identifiers have a 16-bit prefix of leading zeroes, leaving the remaining 16 bits for enumerating bins.

Notably, the *User* need not know *B* ahead of time, as that information is recoverable from the number of leaves in the key tree, itself part of the signed tree head for the key tree (see Section 5.1 for a description of the tree head format). This also permits *Service Provider* to change bin size or number of bins without prior coordination with *Users*, as the *User* will compute their mapping to the key tree when *Service Provider* publishes it.

The leaf nodes in the key tree provide a stable, secret place to house the randomized bin assignment for each *User*. As the *Service Provider* randomizes bin assignment each time they publish their set of canaries, *User* needs a known location to investigate to find their new bin assignment. Randomization of bin assignments is necessary to avoid artificially restricting the search space, and thus the anonymity measure, for a given canary. While strictly speaking a canary has at most entropy proportional to $\frac{1}{N}$ where N is the number of canaries in its bin, we realize an additional entropy advantage as the placement of a canary into a bin is unknown by an adversary.

In so doing we attempt to recover a system-wide maximal entropy of $\frac{1}{U}$ where U is the total number of canaries published by the *Service Provider*, or O(1 billion) in the case of our largest *Service Providers*. This represents a theoretical strengthening of an individual canary's anonymity by four orders of magnitude.

As *User* and *Service Provider* are the only ones in possession of the *User*-submitted salt, the bin identifier is secure. Because the nonces in Section 4.2 need not be computed until the canaries are being populated, *Service Provider* does not persist the mapping of canary to *User*, nor *User* to bin. The key tree mapping for a *User* remains stable until their Canary Locker submits a new salt, which can be done at any time by submitting an appropriate salt pinning message. We recommend that salts rotate regularly, approximately once per publication period.

4.8 Salt Pin Messages

Salt pin messages consist of the private key encrypted concatenation of the first 16-bytes of *User's* username and a client-specified cryptographically random salt. When *Service Provider* decrypts the message with *User's* public key, it checks for the username to ensure validity. *Service Provider* can validate *User's* public key by checking a *Key Service* to find an identity proof for *User's* ownership of the keypair.

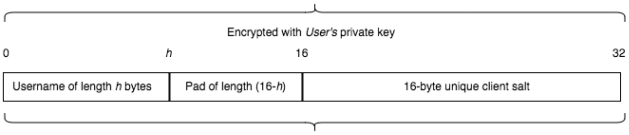

Figure 4. Salt Pin Message format

In this section we detail two attack classes focused on salt pins, a naïve implementation that permits exploitation of these attacks, and our solution.

Given our passive pervasive listener with ad-hoc active capabilities, two attack vectors stand out. First, if salt pin exchanges can be easily fingerprinted, they may be harvested from long term recordings for post-hoc analysis, or actively intercepted and modified or degraded. Second, an active government-level attacker may seek to down a *Service Provider's* canary system by DDoS. We note that the distributed nature of the canary and key tree distribution mechanism (explored in full in Section 5, "Distributed Merkle Tree Format") preclude simple methods for a DDoS to render canary information unreachable. However, salt pin servers must be DDoS-impervious as well, or *Users* will be unable to locate the appropriate canary bin without, in the worst case, downloading ~32GB of data per publication event for an O(1 billion) *Service Provider*.

We adopt a "defense in depth" approach to mitigating these attacks. We note that in accordance with our desire to recover a maximal entropy distribution of $\frac{1}{U}$, network accesses to communicate salt pins must be obfuscated. Thus, in both the naïve solution and our proposed modification, we require Canary Lockers and *Service Provider* salt servers to communicate over an anonymizing mixnet, such that key tree information – which corresponds to identifying the *User's* bin – is not easily tied to an IP address.

[1] "The 🔑 is to have every 🔑" [87].

We note that use of an anonymizing mixnet is also required for distributed lookup and fulfillment of canary and key tree distribution, as detailed in Section 5. We recommend the mixnet i2p [32], noting in Section 9 Appendix A that it offers several advantages for our threat model and system architecture over Tor, chief among them its capability for handling large peer-to-peer traffic.

The naïve solution to processing salt pins is to send them directly over i2p. For example, having a *User* directly submit an encrypted salt pin, along with their username and public key, to *salt.providername.i2p*. This approach is particularly vulnerable to a targeted DDoS attack, as the server must decrypt the salt pin to ensure the username matches (thus establishing ownership over the keypair), and possibly query a third-party *Key Service* to evaluate an identity proof tying the keypair to the *User*. Public key decryption is CPU intensive, and a remote network access consumes an open socket for the duration of the round trip, magnifying the utility of a DDoS attack on the salt server. An attacker could conceivably issue many such requests as a reflection attack on the *Key Server's* infrastructure as well.

Salt servers could require *User* authentication (e.g. using TLS over i2p) before permitting a salt pin message. In this case, the Canary Locker might have to request the *User* to enter their login credentials each time a salt rotation is required. This creates poor security education as *Users* are conditioned to enter their credentials into a login page that is not clearly related to or hosted by their *Service Provider*, providing a lowered threshold for credentials exfiltration. Alternatively, the Canary Locker could perform a kind of "shadow login" to the *Service Provider*, making use of login credentials stored in the browser's password manager.

In this scenario the *User* would only have to supply their private key encrypted salt pin message to the salt server, as it would already know the username to match against and, by virtue of authentication, know which keypair identity proof to request from the *Key Service*. However, this design requires that *Users* store account credentials in the browser's password manager, which may not be the case. Absent this assumption this scenario reverts to the poor security conditioning of the first scenario.

Even assuming all *Users* store their credentials in the browser password manager, this design may mix operational security burdens in a way that makes both *Users* and *Service Providers* uncomfortable. *Users* would be required to trust an additional browser component, the Canary Locker, with login credentials that are normally only under the purview of the password manager. *Service Providers* would have to expose a login listener over an unfamiliar network architecture that aims to produce near-untraceable communications, thereby opening their login infrastructure to sustained and difficult to trace DDoS.

Our solution solves both problems, allowing only authenticated *Users* to submit salts and permitting internal separation of security concerns for both *Users* and *Service Providers*.

We recommend implementing a bootstrapping phase for salt pin messages as a TLS extension. In this method, when *User* navigates to a known *Service Provider* login page, the *User's* web browser would query its Canary Locker to determine the TTL (time to live) of the salt associated with that *Service Provider*. If the salt is valid no action would be taken and the TLS handshake would proceed normally. If the salt is expired, the *ClientHello* message would be modified to include an *AviarySaltPin* extension per RFC 5246 Section 7.4.1.4, "Hello Extensions" [50]. We recommend that *Service Providers* require *Users* to login again when receiving an *AviarySaltPin* in the *ClientHello* on TLS session resumption requests, thereby requiring the initiation of a new TLS session (and protecting against session identifier exfiltration attacks).

We note that the browser must cache an expired salt until two conditions have both been met: it has received an ACK from *Service Provider* for the new salt rotation, and the next round of canaries that post-date the ACK has been pulled. This is necessary because a *Service Provider* may have already prepared a round of upcoming canaries before receiving the new salt. Accordingly, we recommend that Canary Lockers maintain *current_salt* and *previous_salt* values, querying the sub-tree related to the *current_salt* for its canary first, and then querying the bin associated with the *previous_salt* value if the canary is not found initially. If the canary is not found at both locations, it should be considered tripped.

To maintain separation of operational security concerns while decreasing the attack effectiveness of a DDoS on the salt pin servers, we propose the following scheme. On successful handling of the *AviarySaltPin* enhanced *ClientHello* during a login to a given *Service Provider*, the browser passes an encrypted copy of the 48-byte TLS *master_secret* to the Canary Locker. This copy, which we call the "masterhash," is encrypted with the *User's* public key, and thus never leaves the TLS space in plain form. Simultaneously, the *Service Provider* adds a route at their salt server corresponding to the masterhash, for example: salt.providername.i2p/masterhash. *Service Provider* additionally stores the masterhash in a tuple <username, masterhash> at the salt server.

User then posts their salt pin message to salt.serviceprovider.i2p/masterhash, over i2p, via the Canary Locker. The *Service Provider* then uses *User's* public key to decrypt the salt pin message, checks to see the first 16 bytes of the username match the associated *User* in the <username, masterhash> tuple, and then pins the client salt for use in the binning algorithm. Because the *masterhash* is encrypted with the *User's* public key, no bits of the shared TLS *master_secret* are recoverable from it. Furthermore, because peers participating in routing in i2p only decrypt next-hop information, and because the salt server is hosted internal to i2p, the *masterhash* value is never revealed to anyone but *User* and *Service Provider*.

Note that we should only consider this *master_secret* to be safe to use after the handshake messages have been authenticated.[2] Note further that the *Service Provider's* salt server should *not* honor the salt.serviceprovider.i2p/masterhash route *unless* the *User* successfully finishes the login process and authenticates. The TLS *master_secret* persists for the lifetime of the TLS session and exists regardless of key exchange algorithm (e.g. RSA or Diffie-Hellman) used (see Section 8.1 "Computing the Master Secret" of [50]). Accordingly, it is readily available and robust across TLS implementations.

The salt server discards any messages that are not posted to a valid /masterhash route, greatly decreasing the utility of a "spray and

[2] "Designers and implementers should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions," from Section 7.4.1.4, "Hello Extensions." See also, "Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker," from Section 7.3 "Handshake Protocol Overview" [50].

pray" DDoS attack. Routes and tuples can be flushed regularly and relatively quickly, say O(5 min), to avoid the need for large caches. *Service Providers* may communicate /masterhash routes and tuples to salt servers over direct TLS-encrypted persistent TCP connections instead of routing through i2p to avoid added latency and convergence times, while Canary Lockers implement a mandatory 60 second wait-time after receiving an authenticated *masterhash* before posting their salt pin messages via i2p. We note that these connections *must* be properly encrypted, preferably in a post-quantum secure fashion, given NSA's MUSCULAR program, which specifically targets cables connecting the datacenter-datacenter networks of major providers for persistent taps [51].

Since the *masterhash* value could only be known to an authenticated *User*, and could only come from a *Service Provider* who passes a successful TLS certificate, both *User* and *Service Provider* benefit from traditional authentication mechanisms without mixing security burdens with untrusted and untested new components or endpoints.

The naïve solution is also sub-optimal against an ad-hoc active attacker impersonating a *User*. Consider an attacker who mints a new public/private keypair, using their private key to encrypt a salt pin message. In the naïve scenario the attacker could send this message along with the public key, and the salt server would happily decrypt it and find the associated username. If the salt server declined to query the *Key Service* to verify the public key's association with the *User*, or if the attacker had a specific persistent threat spoofing results on the *Key Service* for this *User* in particular, then the attacker could successfully pin a salt that the *User* is unaware of, thus making it much harder for the *User* to know their key leaf and thus find their canary. In our scheme even if the salt server is lazily verifying the *User* and the attacker has successfully compromised the *Key Server*, the attacker would need to additionally impersonate the *User* to the *Service Provider* for login authentication in order to generate a valid *masterhash*. Our solution adds an extra layer of defensive depth, in the form of login credentials, to a successful execution of this attack.

We note that an attacker compromising the *User's* login credentials would be able to rotate their salt at will, rendering Aviary unusable for that *User*. To some extent this outcome renders a warrant canary meaningless, as the adversary would already have access to *User's* data. One possible extension to mitigate this vector would require a *User* rotating a salt to provide in the first field of the salt pin message the 16 bytes of their previous salt, as opposed to their username, which *Service Provider* could then validate before ACK-ing and applying the rotation. In this fashion, valid salt rotation would require the control not only of account credentials and the attested public key, but also control of (or content exfiltration of) the Canary Locker.

We note as well that passing salt messages, as well as canary and key tree retrieval, through i2p frustrates traditional passive fingerprinting methods. In the case of salt pins in particular, there are no message headers to identify this traffic as unique from traditional i2p traffic, and even the destination is ad-hoc, temporary, and known only to *Service Provider* and *User*. As salt pins are a crucial aspect to making Aviary a tractable solution for clients, preventing their passive fingerprinting is particularly useful. See Section 9, "Appendix A: Choosing i2p over Tor" for more.

Finally, we note that the *AviarySaltPin* extension to the *User's Client Hello* message permits post-hoc fingerprinting of recorded data. Canary Lockers could randomly embed *AviarySaltPin* requests in their *ClientHello* messages, even when the TTL of the *current_salt* is still valid, in order to frustrate post-hoc analysis. By simply not submitting a new salt to the salt.providername.i2p/masterhash route, the *User* can avoid registering unnecessary salts and obfuscate which *AviarySaltPin* messages are legitimate.

4.9 Client Keys

Users must provide *Service Providers* a public key for which they own the private key, a classic example of the key distribution problem. We note an elegant solution that meets our requirements for usability in brief, and proceed to elaborate two passive surveillance attacks that *Service Providers* implementing Aviary must guard against.

As the vast majority of *Users* do not possess sufficient security literacy to manage keys themselves, key distribution must be automated by a third-party *Key Service* providing attestation services. At no point should private keys leave the device of a *User*.

Remote *Key Services* like keybase.io can simply the problems of key generation and identity attestation [52]. This third-party service maintains a database where *Users* claim ownership of public keys. These ownership claims are buttressed by identity claims, where a *User* identifies themselves with signed posts on keybase.io and on another service simultaneously, e.g. Twitter.

Keybase.io additionally submits the root of their own Merkle tree (which manages the signed identity statements among other attestation documents managed by their service) into the Bitcoin blockchain, providing further safeguards against compromised servers promoting fake identity claims.

A *Service Provider* should implement two defenses against leaking information in its queries to a third-party *Key Service*. First, it should flip a randomly weighted coin each time it uses the *User's* public key to determine if it should query the *Key Service* to re-authenticate ownership of the key. For example, draw some k from the interval [0, 1], and then draw some y also from [0, 1]. If $y \geq k$, submit a request to the *Key Service* for the latest ownership information.

Second, even after a *Service Provider* has declared a given *User's* warrant canary dead, it should still submit requests for *User's* key ownership information to the *Key Service* to avoid leaking the fact that *User's* warrant canary has triggered to a third party network listener. Accordingly, *Users* without live canaries should be joined to a group from which the *Service Provider* randomly polls the *Key Service* with each publication update. In this fashion neither the *Key Service* nor a passive listener on the network can distinguish between authentic and inauthentic requests to the *Key Service* for client key authentication.

If the *User* wishes to rotate their public/private keypair, they must register the rotation with the *Key Service* in order to provide a valid identity proof for the new keypair, and must send a message (via Canary Locker) to the appropriate *Service Providers* to notify them of the change. We do not specify this message exchange explicitly.

5. DISTRIBUTED MERKLE TREE FORMAT

In this section we describe distributed, co-hosted operation of Aviary canary trees.

5.1 Tree Publications

Service Providers publish three trees each publication event: canary, key, and record. Each tree includes a *signed tree head*, a signed structure enclosing the Merkle hash of the tree root, a date range for which the tree is valid, and the size in number of leaves

of the tree. Signed tree heads and tree data along with the associated magnet/.torrent files constitute a *package*, described in Section 5.2.5. We now describe the third tree, the tree of record.

In order to preserve contiguity between archived trees and currently hosted trees, *Service Providers* should maintain a third Merkle tree, the "tree of record," that records the root nodes of their canary and key trees. As this tree only grows at the rate of three nodes (the signed root nodes for the canary and key trees, along with their signed parent hash) per publication event, it does not contribute a significant storage burden. Note that the individual canary and key tree roots should still be signed by the *Service Provider*.

5.2 Distributed Tree Hosting

There are at least two strategies for hosting the canary trees: traditional servers or peer-to-peer. We note that unlike Bitcoin [53], there exists no decentralized incentive for peers to maintain canary tree data, as there are no coins to mine.[3] On the other hand, centralized hosting places cost directly on the *Service Providers*, and may present a brittle target for State-level DDoS.

We propose an Aviary DHT similar to the BitTorrent Mainline DHT. The Aviary DHT is a BitTorrent DHT with Peer Exchange (PEX) enabled, running over i2p, with an additional announce mechanism in which peers advertise signed magnet links to the latest Aviary trees available. Further nuances are explored below.

In this section we consider three classes of attacks against the Aviary DHT and detail our approach to solving them, along with hardening the DHT against DDoS. We are indebted to the excellent paper by Urdaneta et al for their comprehensive overview [54].

5.2.1 Mitigating Sybil Attacks

Sybil attacks involve an adversary registering a sufficient number of false nodes in the DHT, either to disrupt communications (e.g. by refusing to return key query results or returning junk data, or by poisoning honest nodes' routing tables as in Eclipse attacks) or to manipulate them (e.g. by gaining control of a specific key k and returning subverted data for it). Traditional verification mechanisms [55] require hashing node identifiers on the basis of an IP address. As our *Users* will participate in the DHT over i2p, it is i2p Destinations rather than IP addresses that would need to be hashed. However, since i2p aims to be pseudonymous, provides no limit on the number of Destinations an adversary could populate, and rotates Destinations regularly, such a defense is unlikely to succeed. For the same reason, we discard approaches like the otherwise promising *Peruze*, in which suspicious Sybil nodes are identified by programmatically scanning the DHT to find nodes with high numbers (thousands) of associated IP's [56].

Castro et al. [57] propose a centralized registration authority that creates signed certificates binding a random node identifier to a public key that corresponds to that node, along with its IP address. In their solution a node wishing to register with the DHT mints a public/private keypair and proves ownership of the keypair by unwrapping a server-provided nonce. In return, the registration server creates a signed certificate attesting ownership of the public key to the IP address and the random node identifier, tying the cryptographic identity to a single routable destination within the DHT (node identifier) and a routable destination on the internet (IP address).

Aviary can utilize a similar system with slight modifications, noting again that any *Service Provider* signing certificate used in Aviary should be present in a valid Certificate Transparency log per RFC 6962. Recall that valid nodes have a client salt known only to the *User* and *Service Provider*. *Users* also have public keys with attestation proofs, per the third-party *Key Service*. To register it is sufficient for a *User* to submit a request enclosing their i2p Destination[4] to register.serviceprovider.i2p/route, where *route* == *UPK(*client salt). In so doing, *Users* prove identity to the *Service Provider*, while *Service Providers'* registration servers maintain some level of DDoS protection as they can discard any message sent to an invalid route (Destination rotation is addressed in 5.2.3).

We note that in order to avoid an adversary trawling all possible /route extensions, the *Service Provider* should require some validation beyond knowledge of the /route extension (otherwise, a sufficiently resourced adversary could register false Destinations by trying random routes). This validation could take the form of providing the first 16 bytes of the username associated with the client salt found at *UPK(*client salt*)*. Much like the salt pin, this message should be encrypted with the *User's* private key. We note that registration servers should avoid sending any NACK's in the event of incorrect routes or packets failing validation sent to legitimate routes. These serve only to identify valid DDoS targets for adversaries.

We discard other Sybil mitigation methods as impractical for our deployment scenario, as they rely either on out-of-band social graph information (SybilGuard [58], SybilLimit [59] [60]) or network topology measurements (*netprint* [61]) that will be unreliable under an anonymizing mixnet like i2p [62] [63]. We note that computational puzzles as proposed in [64] [65] may optionally be included to increase the computational cost for systemic perversion of the DHT, at the expense of potentially foreclosing mobile-only *Users* from meaningful participation, as battery drainage likely increases under these conditions. We also note that the hierarchical verification in [65] exposes a new attack vector, in which an adversary seeks to force nodes higher in the hierarchy to lose network reachability. Disgraceful exits cause all child nodes to re-authenticate, potentially providing high-payoff disruption for the cost of isolating or DDoS-ing a small set of *User* nodes, as opposed to higher-resourced *Service Provider* infrastructure. For these reasons we prefer the authenticated registration mechanism above.

5.2.2 Mitigating Eclipse Attacks

In an Eclipse attack an adversary attempts to poison the routes of a specific node in the DHT by occupying the nodes proximal to it. Such an attack may occur on Aviary if an adversary is targeting a *User* of interest in order to degrade their ability to retrieve their warrant canary. Following [57], with DHT's employing proximity metrics it is sufficient to create two routing tables, a trusted and non-optimized table, and an untrusted but route-optimized table.

The untrusted table assumes the proximity metric of the DHT is not compromised. In case of failures (bad data, unreachable routes, etc.) incurred in the use of the optimized table, the trusted table forms a non-optimized but correct fallback. The worst-case scenario involves a fully poisoned optimized table, in which a constant overhead is imposed on all lookup operations as they try and fail the optimized table before falling back to the verified table. Sufficiently intelligent client software could detect when an

[3] Implementation of a blockchain-derived coin architected to meet the needs of Aviary is an interesting strategy the author is presently considering.

[4] This client Destination would be the application-specific Destination the *User* advertises in a LeaseSet in the network database, e.g., the Destination(s) of their i2p BitTorrent client. See https://geti2p.net/en/docs/how/network-database for more.

optimized table is failing more often than not, and react appropriately to avoid this result.

We note that given the strong protection against Sybil attacks noted above, an adversary interested in perverting the Aviary DHT would need to control a number of fake accounts with the *Service Provider* equal to the number of malicious nodes required for their attack. We note that in the instance that all Aviary nodes join a single DHT connecting *Users* of multiple *Service Providers* (in which case nodes would accept results from each other if a signed certificate from *any* of the *Service Providers* is present), the difficulty of this vector reduces to the weakest signup mechanism in place. A simple workaround would be for *Service Providers* to require user accounts to have some minimum length of use before being eligible for inclusion in the warrant canary scheme. Fingerprinting-over-time could also be used to distinguish between true use and programmatically generated accounts, though this is outside the scope of this paper.

5.2.3 Mitigating Routing and Storage Attacks

Finally, we consider routing or storage attacks, in which data distribution within a DHT is disrupted. In these attacks malicious nodes refuse to serve content or serve junk data. We note that the effectiveness of such attacks is strictly gated by the ability of an adversary to join malicious nodes to the DHT. Given our authentication procedures, the difficulty of such an attack is equivalent to the weakness of the *User* sign-up procedure for the weakest *Service Provider* whose *Users* are party to the swarm. Otherwise, BitTorrent already provides data integrity checks (including its own proposed Merkle tree implementation [66]) and provides widespread data replication by design.

We note that i2p presents a unique challenge for the registration mechanism, as i2p Destinations rotate every 10 minutes. Accordingly, we recommend that *Service Providers* actually issue time-stamped signing certificates to nodes that pass registration. Nodes would then use these signing certificates to re-sign new Destinations. Expirations could be enforced directly, in which case an expired signing certificate would cause peers in the DHT to reject the expired node until it produced a new certificate. Distributed enforcement could also occur with computational pressure, in which case peers of an expired node might challenge it with increasing frequency as the time past expiration increases, fielding computational puzzles for the expired node to solve before serving its requests. Such a scenario (along with relaxed frequencies when applicable) might be desirable in the event that *Service Provider* registration servers are downed by an adversary's attack, a fact participating nodes in the DHT can verify independently. In this manner, the swarm could flexibly self-enforce until the registration authorities are able to resume operation.

We note that use of a signing certificate permits tracking of a node's i2p Destinations across time, as the expiration time of the certificate is likely to be longer than the 10-minute Destination rotation schedule of i2p. Canary Lockers can react intelligently to limit this fingerprinting by registering new signing certificates from distinct *Service Providers* in a round robin fashion. Alternatively, re-registration can simply be required for each new Destination, in which case *Service Provider* registration infrastructure represents a critical element for continued operation of the DHT. However, given the relatively small amounts of data a *User* must download, we do not anticipate download of a single *Service Provider's* canary data to require more than one to two i2p Destination rotations (see Section 7.2, "Total Client Cost"). Accordingly, a strong re-registration policy in combination with round-robin registration policy (with randomized round robin order to avoid repeated cycles of certificate rotation) on the client should suffice.

5.2.4 Mitigating Passive Fingerprinting

Running Canary Locker operations as a jailed i2p sub-engine of the browser solves multiple passive fingerprinting strategies, provided that the implementation rejects leaks to non-.i2p domains. This guarantee may be achieved by enforcing a policy that the Canary Locker auto-kills any connection request it would send that is not a request over .i2p. Likewise, the process implementing the Canary Locker should not accept connections from any non-i2p origin.

Accordingly, per RFC 7624 Section 3.1 "Information Subject to Direct Observation," we avoid the following attacks: DNS leaks, cookie tracking across rotated IP's, ISP collusion with governments to identify IP address ownership (routing in i2p is based on the severing the relationship between an IP address and routable destination within the mixnet), username & IP address correlation (where an observer infers a username belongs to an IP by viewing unencrypted IMAP, POP3, SMTP, or SIP traffic), mixed-element HTTP/HTTPS de-anonymization where an HTTPS cookie exposed over HTTP ties a username to HTTP traffic, and TLS session identifier leaks (note our proviso in Section 4.8, "Salt Pin Messages", regarding proper handling of *AviarySaltPin* requests on TLS session resumption).

5.2.5 Hardening Aviary DHT against DDoS

Given the above DHT-specific defenses, we must now consider a State-level DDoS against the centralized portion of the distribution mechanism, which is the *Service Provider* servers hosting .torrent files (or publishing their magnet links) for canary trees, as well as their tree of record and key trees.

We specify the following scheme: a *package* should contain the key tree, canary tree, and tree of record for a given publication event from a given *Service Provider*. The *User's* client software should first download the key tree and tree of record, which can be accomplished by downloading the .torrent file for the *package* and automatically de-selecting all other files for download, thereby ignoring them. After verifying the signed tree head of the key tree with the tree of record, and verifying the signatures with a Certificate Transparency log,[5] the *User* can lookup their bin per the appropriate key tree leaf. At this point the *User* can re-select for download the file in the torrent corresponding to the appropriate canary bin and download only their own bin. Note that the file corresponding to a canary bin must include the co-path from bin root to signed tree head, otherwise client verification of leaf nodes in the bin is impossible. The file corresponding to a key tree leaf must similarly contain co-path from leaf to key tree root.

Our first defense against State-level DDoS on the *Service Provider* distribution infrastructure is the completeness of the *package*. The heartbeat (e.g. "stabilize" function in Chord [67]) of the Aviary DHT can be modified to include the latest known *package* magnet

[5] We note that Certificate Transparency log lookups should be performed against a log hosted on i2p, so as to avoid leaking certificate verification requests to a passive network listener, which would de-anonymize which *Service Provider* a *User* is interested in verifying a canary for, as well as the time at which the request is sent. Such information, combined with statistical analysis on DHT lookup requests over i2p, might de-anonymize the *User* on i2p, permitting linking their i2p identity and IP address.

links for all *packages* known to that peer, or its "manifest."[6] Peers are verified given their possession of a *Service Provider*-issued certificate, which the receiving peer can validate, and thus received *manifests* are assumed correct if the sending peer passes validation.

Even if *manifests* are incorrect, invalid data distribution is impossible due to the signed tree heads in the Merkle trees each *Service Provider* distributes. The risk of "manifest poisoning" is not distinct from the poisoning attacks considered above, and our mitigation strategy is the same. We note that a peer should promiscuously add *package* magnet links to its *manifest* as it learns them to avoid correlating a peer identity with a set of *Service Providers* to which that *User* belongs.

Additionally, deploying over i2p provides non-negligible protection against DDoS attacks-for-hire. DDoS-for-hire relies on botnet infrastructure comprised of thousands to millions of compromised hosts in the wild. These hosts tend to be of two types, either end-user devices such as laptops compromised through advertising network-injected malware, or, compromised "headless" devices and their ilk, such as the Mirai IoT botnet.

In both cases, though the argument is particularly acute in the Mirai case, botnet infrastructure is not easily redeployed to attack a target over i2p. Attacking an i2p target requires the host systems to have i2p installed and running. While some malware certainly could be repurposed to include an i2p service as part of its payload, this represents a significant engineering effort and in the case of IoT devices may simply be unworkable without extensive engineering efforts.

These factors conspire to increase the "non-recoverable engineering" (NRE) costs of a DDoS on Aviary infrastructure, dis-incentivizing market-motivated actors (or, equivalently, raising the price of a one-off solution). As a result, it is likely that the decision to deploy over i2p may help narrow the set of likely DDoS attackers to State-level actors only, who may find it difficult to mask their operations without the cover of a criminal botnet. In turn, this decreased cover raises the likelihood that the State actor is identified, increasing the risk of undertaking an attack.

Finally, *Service Providers* should implement i2p multihoming, in which multiple servers can co-host a single hidden service (or *eepsite*), in conjunction with an appropriate load balancer [92]. In the event that a DDoS attack is detected, e.g. on a salt server or DHT registration server, the *Service Provider* can spin up additional i2p routers and multihome them on-demand, letting the load balancer redistribute the incoming requests. In the event that service is interrupted distribution should continue for *Users* who have already passed DHT registration, with registration returning once the DDoS subsides. Note as well that peers can validate each other's certificates in the absence of functioning *Service Provider* i2p infrastructure, so long as a Certificate Transparency log containing the *Service Provider's* signing certificate remains reachable, and might decide to relax the frequency of their computational puzzle challenges in cases of extended outages.

5.3 Publishing New Packages

When a *Service Provider* is ready to publish a magnet link for a new package it signs the magnet link and publishes it via its *Tree Hosts* to all known peers in the DHT. When a peer receives a new magnet link it verifies the signature and begins propagating the signed link via its heartbeat messages as part of its manifest. The key advantage of a magnet link is that it avoids the need for a centralized repository of .torrent files, enabling peers to directly bootstrap a swarm around a new file [88].

Several strategies are available to bootstrap initial announcements of new packages: *Tree Hosts* may optionally maintain larger peer lists so that their initial announce reaches more peers; responses to the BitTorrent DHT's *get_peers* request can be modified to include the peer's Aviary *manifest* (list of *packages*) [68]; and as discussed previously, nodes in the Aviary DHT announce new packages to each other via heartbeat. Note that we avoid BEP 44's extension for storing mutable data as a format for announcing new trees, as it A) requires the node in the DHT listing the entry to be honest and B) provides no simple way for old tree publications to be indexed by magnet link [69].

New packages and their magnet links should also be made available on the *Service Provider's* "archive of longest and last resort," explained in the following section. Finally, *Users'* requests for key trees (and subsequently their canary bins) should be staggered so as not to overwhelm Tree Hosts with a distributed thundering herd. Each Canary Locker might add a random fuzz factor of up to 24 hours to the expiration date of the most recent canary root node in the User's possession, so as to allow both *Tree Hosts* and peers to propagate data throughout the swarm.

5.4 Cooperative Hosting and Archiving

Service Providers should operate *Tree Hosts* and co-host each other's Aviary trees. Volunteers such as academic institutions, libraries, maker and hacker-spaces, and individuals can also operate independent *Tree Hosts*. Merkle tree root signatures protect distributed tree material from malicious hosts, as tampering is facilely evident to any *User*, and *Service Provider*-signed magnet links provide checksums for tree material via BitTorrent. A *Tree Host* is any host seeding a full copy of one or more *Service Provider's* "packages," as defined above, including previously published copies.

Our recommendation is for server-side delivery mechanisms (HTTPS, FTPS) to operate as archives of "last and longest resort." These direct connections can facilitate new *Tree Hosts* who desire to join the BitTorrent swarm as a full seed, as well as providing a last resort for peers who cannot complete their downloads from the swarm due to poor peer availability. Long-term maintenance of these archival hosts is best performed by organizations with funding and incentive to maintain, e.g. *Service Providers*. Direct i2p URL's may be included within the Canary Locker as a hardcoded bootstrapping mechanism, with throttling on the HTTPS/FTPS delivery mechanism to protect *Service Provider* resources. In the event of a viable DHT, such resources are expected to be minimally used.

As trees are large, O(32GB) uncompressed for an O(1 billion) *Service Provider*, we recommend a two-tier archiving strategy. *Tree Hosts* should expect to cache at least the previous 3 months of tree publications. Trees older than 3 months should be published with The Internet Archive, a long-term archival project aimed at archiving the internet (which is presently building a Canadian branch, specifically for the purposes of combatting US internet surveillance [70] [89]). *Service Providers* should contribute modest recurring funds towards the provisioning of additional storage for this purpose. Alternatively, as canary and key trees older than 12 months are almost certainly useless, they can be deleted. *Users* interested in preserving their own canaries for evidentiary reasons

[6] Note that "manifest" is used here in a sense independent from its meaning as a "manifest topic" in the Magnet URI scheme.

can preserve key leaves with co-path and signed key root, along with their individual canary and its co-path to the signed canary root. This collection together with the tree of record constitutes authenticated, non-repudiable evidence of a canary publication for the *User* from the *Service Provider*.

6. OBFUSCATING AVIARY CANARY COUNTS WITH HIDDEN NUMBERS

We describe a scheme for *Service Providers* to publish Aviary canaries to the distributed Merkle tree without revealing precise canary counts. We first explore a naïve implementation that does not work, refining it to meet both legal and competitive pressures.

6.1 Motivating the Hidden Number Problem

Per Section 4.3, "Dummy Canaries," a *Service Provider* is not permitted to provide an exact count of the number of secret government requests they have fielded for spying on their *Users*, hence the numeric reporting ranges. If the *Service Provider* simply refused to publish a node for each *User* the government was snooping on, this could provide a near-exact count (assuming tight estimates for churn and user growth between tree publications) of the number of secret orders served.

Second, publishing exact user counts repeatedly over time generates competitive knowledge that may be exploitable by corporate adversaries, e.g., granting insight into user growth and churn, seasonality, etc. Thus, for both economic and legal reasons, firms are incentivized to mask the true number of *Users* they are serving, and a successful distributed warrant canary scheme must take this into account. We term this problem the "hidden number problem," as the goal is to obfuscate both number of tripped canaries and number of *Users*.

6.2 Solving the Hidden Number Problem

Given a *Service Provider* with n true *Users*, we desire a publishing strategy that meets the requirements specified in the rest of the paper while obfuscating the number n of true *Users*, as well as the number t of tripped canaries. We note that naively publishing $(n\text{-}t)_i$ canaries for each interval $i \in I$ where I is the set of publication periods, exposes t, per our analysis in Section 6.1. The solution is to introduce t many dummy canaries each publication period, such that the number of nodes published is $(n - t + t) = n$.

We can refine this strategy to publish $(n + \delta)$ many nodes, where δ is some random number of dummy nodes, in order to obfuscate the number n. We suggest that δ be a number drawn at random from the uniform distribution $U(m\_min, m\_max)$, where m_min, m_max are some non-zero $m \in \mathbb{N}^+$, with $m\_max > m\_min$. We further require that a new $U(m\_min, m\_max)$ is specified for each publication event.

We note that n remains recoverable only if an observer has sufficient information to specify the uniform distribution from which δ is drawn. Imagine a *Service Provider* (with an unchanging n) who uses the same uniform distribution across canary tree publication events. An observer can establish a *recorded_min* and *recorded_max*, which respectively are approximately equal to (n + *m_min*) and (n + *m_max*). Subtracting *recorded_min* from *recorded_max* yields the *length* of the uniform distribution. If the observer knows one of the endpoints, then the center of the distribution is recoverable. As the expectation of multiple draws from a uniform random distribution is its center, subtracting the center from each of the recorded canary leaf counts and averaging the result recovers the hidden number n, with greater precision as the number of publication events recorded increases.

Absent the information needed to specify the uniform distribution, the observer is unable to produce the center of the uniform distribution, and thus unable to recover n. Specifying a uniform distribution $U(a, b)$ requires knowing (a AND b) OR (a XOR b AND *length*), where *length* is the length of the distribution. We note that repeated publication with δ drawn from the same uniform distribution exposes *length* after a long enough set of samples, while failing to properly randomize either *m_min* or *m_max* to be a non-zero positive integer may inadvertently leak the center of the distribution, e.g. if $a == 0$, or is otherwise known (or b is), then knowledge of *length* alone is sufficient to recover the center.

Accordingly, we recommend *m_min* and *m_max* be non-zero randomly selected positive integers, and that the range (*m_min*, *m_max*) from which δ is drawn change with each publication event. We note that relying on churn and user growth alone to obfuscate n from publication to publication is insufficient, and in itself may constitute economically competitive knowledge a *Service Provider* seeks to deny their competitors. With the addition of δ as specified, statistical estimates of the user growth and churn exhibited by changes in n over time are frustrated.

7. IMPLEMENTATION CONSIDERATIONS

In the following section, we discuss some implementation considerations for Aviary. Specifically, Merkle tree implementation details and client costs for *Users*.

7.1 Choosing a Cryptographic Hash Function

Merkle trees are binary hash trees whose leaf nodes are composed of hashes of the input data nodes. For Aviary's canary tree, each data node corresponds to a single canary. While this section analyzes canary trees, similar reasoning applies for key and record trees, with their data nodes defined as previously specified.

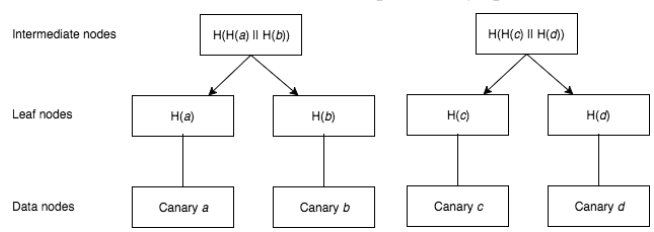

Figure 5. First two tree levels for a naïve Aviary Merkle tree.

In the above diagram, the *data nodes* refer to the individual canaries, *a, b, c,* and *d*. The *leaf nodes* refer to the hashes of the data nodes, H(a), H(b), etc. The intermediate nodes shown are the first non-leaf nodes of the Merkle tree, where H(H(a) || H(b)) is the hash of the concatenation of the hashes of data nodes a and b.

We note that the hash function chosen must be a cryptographically strong hash, which we specify to mean a hash function exhibiting the following properties: pre-image resistance (also known as a "one-way" hash function), second pre-image resistance, and collision resistance. See [71] for a detailed discussion of these properties.

For our purposes we now specify a trivial attack the naïve formulation of a Merkle tree exposes, as well as how to mitigate it. This attack, known as a trivial second pre-image attack on a Merkle tree, involves the inability of an auditor to definitively claim the presence or absence of a given data block within the tree. Note that while an individual cryptographically secure hash function possesses second pre-image resistance, its utilization within a naïve Merkle tree does not extend this property to the tree as a whole.

Consider some data block *e* whose value is defined as H(H(*a*) || H(*b*)). Asked to verify that a hash representing *e* is in the naïve tree above, an auditor finds a valid chain linking our first intermediate node to the root node, and reports *e* is present. Unintended data has now been verified as part of the tree. Or, if the format of the data nodes is such that a data node with *e*'s format is forbidden, an auditor might report that corrupted data is present in the tree. Or, if an auditor is asked to prove that nodes *a* and *b* are in the tree, she will be unable to confirm their presence, as the second pre-image *e* is not ruled out, since the auditor cannot claim that *e* is not in the tree.

Two options present themselves for mitigating this attack. First, the signed tree head at the root of the Merkle tree must include the size of the tree. As the tree is a binary tree, a size (in number of leaf nodes) uniquely specifies the shape of a given tree, even if that tree is unbalanced, provided that trees are filled uniformly (e.g. always from left to right). Given an index and rank, or level, of a candidate node, combined with the size of the tree, one can determine if that node is a leaf or intermediate.

Second, distinct hashing algorithms should be used for computing leaf nodes and computing intermediate nodes. In such a fashion, no possibility for confusing a second pre-image with a valid data node is possible. Though Certificate Transparency uses SHA-256 for all hashes, it implements a similar separation between leaf and intermediate nodes, hashing leaves with an additional 0x00 byte, while intermediate nodes are hashed with an additional 0x01 byte (see Section 2.1, "Merkle Hash Trees", of [35]).

As canary nodes in Aviary are 32 bytes long, many cryptographically secure hash algorithms can process a canary node in one hash invocation. Since canary leaf nodes form the majority of the cryptographic computational workload for a publication set, the fastest hashing algorithm should be used at the leaf level, with the next fastest used above it. We note that some high-performing hash algorithms evaluated cross-platform on 64-byte payloads include SKEIN-512-256 [72], BLAKE-512 [73], and SHA3-512 [74]. Results are available at [75]. A discussion of their relative security trade-offs is outside the scope of this paper.

7.2 Total Client Cost

We note that Canary Lockers may restrict total i2p bandwidth usage to minimize data costs for *Users*. Such a limit may be necessary, as in i2p all peers are also mixnet routers, and engage in network traffic unrelated to an individual *User's* applications. Additionally, the bandwidth speed available to i2p can be tuned to avoid impacting performance of other *User* applications. We note that such tuning should be uniform. Lowering the speed available to i2p when not downloading Aviary material may leak timing of canary downloads to an adversary listening on the network, as a significant jump in i2p traffic may occur as speed limits are relaxed.

The current default speed limit is 32KBps, yielding an expected BitTorrent bandwidth of roughly 15KBps [76]. Assuming the default speed and a canary tree of 100,000 32-byte nodes, we note that a *User* would download the 3.2MB of tree data in roughly three and a half minutes. While this estimate does not include the overhead introduced by encrypting the canary nodes, as this overhead varies with selection of encryption algorithms, it should serve as a rough approximation. Key trees (~400KB) and record trees ($\ll$20KB) do not contribute significantly to the data budget, and compression across the 100,000 node bins may be able to recover some of the overhead introduced by encryption.

8. CONCLUSION

We have presented the first scheme for a global, distributed, confidential, user-friendly per-user warrant canary system. Through a layered approach of distributed Merkle trees, we have increased the anonymity set of a *User* to match the total population for a *Service Provider*. Additionally, we have specified implementation details both client and server side, as well as tactics for hardening Aviary systems against multiple types of attacks, both passive and active, including *User* impersonation and State-level DDoS. Finally, these techniques are orchestrated in a fully automated manner requiring close to zero *User* interaction.

We note that future research on this question should include post-quantum security of Aviary canaries, with "hybrid forward secrecy" providing one option [77]. Indeed, Google is already experimenting with NewHope implementations within Chrome [78], and NIST is sponsoring an upcoming post-quantum competition in 2017 [79].

Readers may rightly opine that implementing such a system will not be simple, but defending global civil society from the world's most powerful governments is not a trivial task. Fortunately, technology *Service Providers* possess both the political will and technical expertise to realize such a system, and additional complexity reductions may reveal themselves upon further study.

[x – Insert Acknowledgements]

9. Appendix A: Choosing i2p over Tor

Given our threat model's assumption of a state-level actor with massive pervasive surveillance and ad-hoc active attacks on targeted *Users*, we believe i2p to be a superior choice to Tor.

i2p is architected in a fashion that is peer-to-peer friendly, whereas Tor explicitly suffers from congestion with increased p2p traffic [80] [81], as well as not supporting UDP-based protocols [82], which can be used to cut traffic to a *Service Provider* tracker by roughly half compared to HTTP. It is worth noting no i2p torrent trackers/clients currently support UDP announcements, though differences from BEP standards are published to enable UDP client/tracker development for i2p [83] [84]. This difference is a consequence of design philosophy. i2p as a network provides increased anonymity with increased traffic, and is architected for the benefit of "hidden service" operation, e.g. services hosted within the darknet, whereas Tor attempts to provide a proxy service via mixnet to communicate with the clearnet.

It is also worth noting in [80] above, that *Users* using Tor to browse the web and access a BitTorrent swarm were at increased risk of providing specific cross-correlation of those streams, thereby de-anonymizing not only their BitTorrent traffic but also their web traffic. As Tor is more widely used as a general purpose browser, given the TorBrowser bundle and its stated design goal of browsing the clearnet anonymously, we believe it is best to avoid mixing burdens and deploy i2p for Aviary purposes instead.

Additional reasons to prefer i2p include i2p's decentralized and distributed address resolution mechanism, as opposed to Tor's centralized directory services , which impact DDoS resilience; i2p's use of short-lived unidirectional tunnels, as opposed to Tor's long-lived bidirectional tunnels, which halve the number of nodes an attacker needs to compromise to recover the same information on a message sender or recipient (hardening against targeted ad-hoc active surveillance); and an increased relative anonymity set, as all peers participate in routing in i2p.

Furthermore, while the fact that an internet user is participating in i2p is not hidden, the nature of their participation is, since all peers

act as routers, carrying several message and traffic types that are not distinguishable. Thus, passive fingerprinting to disentangle "i2p usage" from "i2p usage for Aviary" is frustrated.

The primary technical challenge for i2p relates to scaling the network database backend to support a massive increase in users, as would occur under a successful Aviary deployment. The i2p dev team has already specified bounties for this work, with two of the three preliminary bounties being fulfilled [85]. Moreover, scaling distributed systems is exactly the kind of expertise the tech industry is well-positioned to provide. A more in-depth comparison between Tor and i2p may be viewed at [86].

10. REFERENCES

[1] Opsahl, Kurt. 2014. "Warrant Canary Frequently Asked Questions." *Electronic Frontier Foundation*. April 10, 2014. https://www.eff.org/deeplinks/2014/04/warrant-canary-faq.

[2] Farivar, Cyrus. 2016. " Reddit removes 'warrant canary' from its latest transparency report." *Ars Technica*. March 31, 2016. http://arstechnica.com/tech-policy/2016/03/reddit-removes-warrant-canary-from-its-latest-transparency-report/.

[3] Kaufman, Brett. 2015. "US Government Makes Slight Concession in Twitter's Warrant-Canary Suit." *Just Security*. March 9, 2015. https://www.justsecurity.org/20850/government-slight-concession-twitters-warrant-canary-suit/.

[4] Levison, Ladar. 2013. "My Fellow Users." *Lavabit.com*. August 8, 2013. https://lavabit.com/.

[5] Canary Watch. 2016. "Frequently Asked Questions." *Electronic Frontier Foundation, Berkman Center* et al. May 25, 2016. https://canarywatch.org/faq.html.

[6] "Deputy Attorney General Transparency Reporting Letter." 2014. Letter to Facebook, Google, LinkedIn, Microsoft, and Yahoo! outlining how companies can publicly report national security requests. *Electronic Frontier Foundation*. January 27, 2014. https://www.eff.org/document/deputy-attorney-general-transparency-reporting-letter.

[7] Quintin, Cooper. "Canary Watch – One Year Later." 2016. *Electronic Frontier Foundation*. May 25, 2016. https://www.eff.org/deeplinks/2016/05/canary-watch-one-year-later.

[8] Zhu, Yan. 2016. "surveillance, whistleblowing, and security engineering." *DiracDeltas*. October 5, 2016. https://diracdeltas.github.io/blog/surveillance/.

[9] Farrell, S. et al. 2014. RFC 7258 "Pervasive Monitoring Is An Attack." *Internet Engineering Task Force: Best Current Practice*. May 2014. https://tools.ietf.org/html/rfc7258.

[10] Muffett, Alec. 2014. "Making Connections to Facebook more Secure." *Facebook*. October 31, 2014. https://www.facebook.com/notes/protect-the-graph/making-connections-to-facebook-more-secure/1526085754298237/.

[11] Muffett, Alec. 2016. "1 Million People use Facebook over Tor." *Facebook*. April 22, 2016. https://www.facebook.com/notes/facebook-over-tor/1-million-people-use-facebook-over-tor/865624066877648/.

[12] Gagliordi, Natalie. 2016. "Microsoft's Brad Smith: 'The path to hell starts at the backdoor.'" *ZDNet*. March 1, 2016. http://www.zdnet.com/article/microsofts-brad-smith-the-path-to-hell-starts-at-the-backdoor/.

[13] Smith, Brad. 2015. "Our legal challenge to a US government search warrant." *Microsoft On the Issues*. April 9, 2015. https://blogs.microsoft.com/on-the-issues/2015/04/09/our-legal-challenge-to-a-us-government-search-warrant/.

[14] Smith, Brad. 2016. "Written Testimony of Brad Smith President and Chief Legal Officer, Microsoft Corporation," before the House Judiciary Committee Hearing on International Conflicts of Law Concerning Cross Border Data Flow and Law Enforcement Requests. *United States House of Representatives Judiciary Committee*. February 25, 2016. https://judiciary.house.gov/wp-content/uploads/2016/02/brad-smith-testimony.pdf.

[15] Brandom, Russell. " The FBI just got its hands on data that Twitter wouldn't give the CIA." *The Verge*. November 14, 2016. http://www.theverge.com/2016/11/14/13629248/fbi-dataminr-twitter-surveillance-contract-scanning-police.

[16] Cook, Tim. 2016. "A Message to Our Customers." *Apple*. February 16, 2016. https://www.apple.com/customer-letter/.

[17] Tech Wire Asia. 2016. "Google issues warnings to US journalists, professors of 'govt-backed' hack attempts." November 24, 2016. http://techwireasia.com/2016/11/google-warnings-journalists-professors-govt-backed-hack-attempts/.

[18] Ghosh, Agamoni. 2016. "Google sends state-sponsored hack warnings to numerous journalists and professors." *International Business Times*. November 24, 2016. http://www.ibtimes.co.uk/google-sends-state-sponsored-hack-warnings-numerous-journalists-professors-1593172.

[19] Krugman, Paul. 2016. "A number of liberal writers, me included, seem to have gotten this notice yesterday." *@paulkrugman Twitter*. November 23, 2016. https://twitter.com/paulkrugman/status/801473411943923712.

[20] Chait, Jonathan. 2016. "Same. Also, I've been getting emails from what is almost surely a hacker like the kind described here: https://t.co/eYnG8061KY." *@jonathanchait Twitter*. November 22, 2016. https://twitter.com/jonathanchait/status/801232433639133184.

[21] Klein, Ezra. 2016. "Well, this is a scary message to get from Google." *@ezraklein Twitter*. November 23, 2016. https://twitter.com/ezraklein/status/801482484768796672.

[22] Ioffe, Julia. 2016. "And here we are." *@juliaioffe Twitter*. November 23, 2016. https://twitter.com/juliaioffe/status/801435745760186368.

[23] Olbermann, Keith. 2016. "Hacking vote? Trying to hack my email and Krugman's? Hacking Twitter founder? Lines up with Trump demonizing dissent." *@KeithOlbermann Twitter*. November 22, 2016. https://twitter.com/KeithOlbermann/status/801265606121164801.

[24] Franke-Ruta, Garance. 2016. "Google issues warnings to US journalists, professors of 'govt-backed' hack attempts: https://t.co/dN6rbImJ1J @TechWireAsia." *@thegarance Twitter*. November 24, 2016. https://twitter.com/thegarance/status/801799320479694848.

[25] Penney, Jon. 2016. "Chilling Effects: Online Surveillance and Wikipedia Use." *Berkeley Technology Law Journal* Vol. 31, No. 1. 2016.

https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2769645.

[26] Greenwald, Glenn. 2016. "New Study Shows Mass Surveillance Breeds Meekness, Fear and Self-Censorship." *The Intercept*. April 28, 2016. https://theintercept.com/2016/04/28/new-study-shows-mass-surveillance-breeds-meekness-fear-and-self-censorship/.

[27] PEN American Center. 2013. "Chilling Effects: NSA Surveillance Drives U.S. Writers to Self-Censor." *PEN American Center*. November 12, 2013. https://pen.org/sites/default/files/2014-08-01_Full%20Report_Chilling%20Effects%20w%20Color%20cover-UPDATED.pdf.

[28] Gilens, Naomi. 2015. "The NSA has not been here: warrant canaries as tools for transparency in the wake of the Snowden disclosures." *Harvard Journal of Law & Technology* Vol. 28, No. 2. Spring 2015. http://jolt.law.harvard.edu/articles/pdf/v28/28HarvJLTech525.pdf.

[29] Wexler, Rebecca. 2014. "Warrant Canaries and Disclosure by Design: The Real Threat to National Security Letter Gag Orders." *Yale Law Journal* Vol 124. December 19, 2014. http://www.yalelawjournal.org/forum/warrant-canaries-and-disclosure-by-design.

[30] Penney, Jon. 2014. "Warrant Canaries Beyond the First Amendment: A Comment." In *Internet Monitor 2014: Reflections on the Digital World: Platforms, Policy, Privacy, and Public Discourse, Berkman Klein Center for Internet & Society*. Harvard University 49 (2014). Berkman Center Research Publication No. 2014-17. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2869583.

[31] Tor Project, Homepage. Accessed November 2016. https://torproject.org.

[32] The Invisible Internet Project, Homepage. Accessed November 2016. https://geti2p.net.

[33] Tahoe Least Authority File Store, Homepage. Accessed November 2016. https://tahoe-lafs.org.

[34] Reid, Fergal and Harrigan, Martin. 2012. "An Analysis of Anonymity in the Bitcoin System." In *Security and Privacy in Social Networks*. Springer New York, 2013. https://arxiv.org/abs/1107.4524.

[35] Laurie, B. et al. 2013. RFC 6962 "Certificate Transparency." *Internet Engineering Task Force: Experimental*. June 2013. https://tools.ietf.org/html/rfc6962.

[36] Serjantov, Andrei and Danezis, George. 2002. "Towards an Information Theoretic Metric for Anonymity." *Privacy Enhancing Technologies PET '02*. April 14-15 2002. https://www.cs.ucsb.edu/~ravenben/classes/595n-s07/papers/anon-serj.pdf.

[37] Wang, Qiyan and Borisov, Nikita. 2012. "Octopus: A Secure and Anonymous DHT Lookup." https://arxiv.org/abs/1203.2668.

[38] Crosby, Scott A. and Wallach, Dan S. 2009. "Efficient Data Structures for Tamper-Evident Logging." *18th USENIX Security Symposium*. http://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf.

[39] Cisco. No date given. "A Cisco Guide to Defending Against Distributed Denial of Service Attacks." *Cisco Security Research & Operations*. Accessed November 2016. https://www.cisco.com/c/en/us/about/security-center/guide-ddos-defense.html.

[40] Barnes, R. et al. 2015. RFC 7624 "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement." *Internet Architecture Board: Informational*. August 2015. https://tools.ietf.org/html/rfc7624.

[41] Cox, Joseph. 2016. "The FBI Hacked Over 8,000 Computers In 120 Countries Based on One Warrant." *Motherboard (Vice)*. November 22, 2016. https://motherboard.vice.com/read/fbi-hacked-over-8000-computers-in-120-countries-based-on-one-warrant.

[42] Cox, Joseph. 2016. "Unsealed Court Docs Show FBI Used Malware Like 'A Grenade'." *Motherboard (Vice)*. November 7, 2016. https://motherboard.vice.com/read/unsealed-court-docs-show-fbi-used-malware-like-a-grenade.

[43] FBI. 2013. "Affidavit in Support of Application for Search Warrant." *United States District Court for the District of Maryland: In the matter of the search of computers that access "Websites 1-23"*. Case No. 13-1744 WC. July 22, 2013. https://www.documentcloud.org/documents/3215133-Freedom-Hosting-NIT-Affidavit.html.

[44] FBI. 2013. "Affidavit in Support of Application for Search Warrant." *United States District Court for the District of Maryland: In the matter of the search of computers that access the e-mail accounts described in Attachment A*. Case No. 13-1746 WC. https://www.documentcloud.org/documents/3215129-Affidavit-in-TorMail-NIT-Case.html.

[45] Reddit. 2016. "Traffic statistics for /r/AskReddit." *Reddit.com*. Accessed November 2016. https://www.reddit.com/r/AskReddit/about/traffic/.

[46] Schneier, Bruce. 2016. "Reddit's Warrant Canary Just Died." *Schneier on Security*. April 1, 2016. https://www.schneier.com/blog/archives/2016/04/reddits_warrant.html.

[47] Epic.org. No date given. "National Security Letters." *Electronic Privacy Information Center*. Accessed November 2016. https://epic.org/privacy/nsl/.

[48] Collet, Yann. 2016. "xxHash – Extremely fast hash algorithm." *Github*. Accessed November 2016. https://github.com/Cyan4973/xxHash.

[49] Yuriev, Leonid. 2016. "t1ha." *Github*. Accessed November 2016. https://github.com/PositiveTechnologies/t1ha.

[50] Dierks, T. and Rescorla, E. 2008. "The Transport Layer Security (TLS) Protocol Version 1.2." *Network Working Group: Standards Track*. August 2008. https://tools.ietf.org/html/rfc5246.

[51] Rushe, Dominic et al. 2013. "Reports that NSA taps into Google and Yahoo data hubs infuriate tech giants." *The Guardian*. October 31, 2013. https://www.theguardian.com/technology/2013/oct/30/google-reports-nsa-secretly-intercepts-data-links.

[52] Keybase. No date given. Docs > Server Security. *Keybase.io*. Accessed November, 2016. https://keybase.io/docs/server_security.

[53] Nakamoto, Satoshi. 2008. "Bitcoin: A Peer-to-Peer Electronic Cash System." https://bitcoin.org/bitcoin.pdf

[54] Urdaneta et al. 2011. "A Survey of DHT Security Techniques." *ACM Computing Surveys*, Vol. 43 No. 2. June 2011. https://www.distributed-systems.net/papers/2011.acm-cs.pdf.

[55] Dinger, J. and Hartenstein, H. 2006. "Defending the Sybil Attack in P2P Networks: Taxonomy, Challenges, and a Proposal for Self-Registration." In *Proc. 1st International Conference on Availability, Reliability and Security (Vienna, Austria)*. IEEE Computer Society Press, Los Alamitos, CA., 756–763.

[56] Wolchok, Scott et al. 2010. "Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs." In *Proc. 17th Network and Distributed System Security Symposium* (NDSS 2010). February-March 2010. https://jhalderm.com/pub/papers/unvanish-ndss10-web.pdf.

[57] Castro et al. 2002. "Secure Routing for Structured Peer-to-Peer Overlay Networks." In *Proc. 5th Symposium on Operating System Design and Implementation (Boston, MA)*. http://www.cs.rice.edu/~dwallach/pub/osdi2002.pdf

[58] Yu et al. 2006. "SybilGuard: Defending Against Sybil Attacks via Social Networks." In *Proc. SIGCOMM (Pisa, Italy)*. ACM Press, New York, NY, 267–278.

[59] Yu et al. 2008. "SybilLimit: A Near-Optimal Social Network Defense against Sybil Attacks." In *Proc. International Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA., 3–17.

[60] Danezis et al. 2005. "Sybil-Resistant DHT Routing." In *Proc. 10th European Symposium on Research in Computer Security. Lecture Notes on Computer Science.* Springer-Verlag, Berlin, 305–318.

[61] Wang et al. 2005. "An Efficient and Secure Peer-to-Peer Overlay Network." In *Proc. 30th Local Computer Networks.* IEEE Computer Society Press, Los Alamitos, CA., 764–771

[62] Bazzi, R. A. and Konjevod, G. 2005. "On the Establishment of Distinct Identities in Overlay Networks." In *Proc. 24th Symposium on Principles of Distributed Computing (Las Vegas, NV)*. ACM Press, New York, NY,312–320

[63] Bazzi et al. 2006. "Hop Chains: Secure Routing and the Establishment of Distinct Identities." In *Proc. 10th International Conference on Principles of Distributed Systems (Bordeaux, France). Lecture Notes on Computer Science*, Vol. 4305. Springer-Verlag, Berlin, 365–379

[64] Borisov, N. 2006. "Computational Puzzles as Sybil Defenses." In *Proc. 6th International Conference on Peer-to-Peer Computing*. IEEE Computer Society Press, Los Alamitos, CA., 171–176.

[65] Rowaihy, H. et al. 2007. "Limiting Sybil Attacks in Structured Peer-to-Peer Networks." In *Proc. 26th INFOCOM Conference (St. Louis, MO)*. IEEE Computer Society Press, Los Alamitos, CA., 2596–2600.

[66] Bakker, Arno. 2009. BEP 30 "Merkle hash torrent extension." *BitTorrent.org Standards Track*. March 11, 2009. http://bittorrent.org/beps/bep_0030.html.

[67] Stoica, Ion et al. 2001. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications." *SIGCOMM '01*. August 27-31, 2001 (San Diego, California). https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf.

[68] Loewenstern, Andrew and Norberg, Arvid. 2008. BEP 5 "DHT Protocol." *BitTorrent.org Standards Track*. January 31, 2008 (last updated March 22, 2013). http://www.bittorrent.org/beps/bep_0005.html.

[69] Norberg, Arvid and Siloti, Steven. 2014. BEP 44 "Storing arbitrary data in the DHT." December 19, 2014 (last updated August 6, 2016). http://bittorrent.org/beps/bep_0044.html.

[70] Kahle, Brewster. 2016. "Help Us Keep the Archive Free, Accessible, and Reader Private." *Internet Archive Blogs*. November 29, 2016. https://blog.archive.org/2016/11/29/help-us-keep-the-archive-free-accessible-and-private/.

[71] Menezes, Alfred et al. 2001. "Chapter 9: Hash Functions and Data Integrity." *Handbook of Applied Cryptography*. p. 324. http://cacr.uwaterloo.ca/hac/.

[72] Ferguson, Niels et al. No date given. "The Skein Hash Function Family." *Schneier on Security*. Accessed November 2016. https://www.schneier.com/academic/skein/.

[73] Aumasson, Jean-Philippe et al. No date given. "SHA-3 proposal BLAKE." https://131002.net/blake/.

[74] FIPS PUB 202. 2015. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." *National Institute of Standards and Technology*. August 2015. https://dx.doi.org/10.6028/NIST.FIPS.202.

[75] ECRYPT Benchmarking of All Submitted Hashes. 2016. "Measurements of SHA-3 finalists, indexed by machine." *VAMPIRE (Virtual Applications and Implementations Research Lab)*. October 26, 2016 (see watermark on graph image "crypto_sha3 64 bytes"). https://bench.cr.yp.to/results-sha3.html.

[76] The Invisible Internet Project. No date given. "Why is I2P so slow?". *Frequently Asked Questions*. https://geti2p.net/en/faq#slow.

[77] Weatherly, Rhys. 2016. "Noise Extension: Hybrid Forward Secrecy." *Github*. October 8, 2016. https://github.com/rweather/noise_spec/blob/forward_secrecy/extensions/ext_hybrid_forward_secrecy.md.

[78] Braithwaite, Matt. 2016. "Experimenting with Post-Quantum Cryptography." *Google Security Blog*. July 7, 2016. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[79] National Institute of Standards and Technology. 2016. "Proposed Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process." August 1, 2016. http://csrc.nist.gov/groups/ST/post-quantum-crypto/documents/call-for-proposals-draft-aug-2016.pdf.

[80] The Tor Project. 2010. "Bittorrent over Tor isn't a good idea." April 30, 2010. https://blog.torproject.org/blog/bittorrent-over-tor-isnt-good-idea.

[81] The Tor Project. 2009. "Why Tor is slow and what we're going to do about it." March 13, 2009. https://blog.torproject.org/blog/why-tor-is-slow.

[82] proper. 2012. "UDP over Tor." *Tor Bug Tracker & Wiki*. December 30, 2012. https://trac.torproject.org/projects/tor/ticket/7830.

[83] Van der Spek, Olaf. 2008. BEP 15 "UDP Tracker Protocol for BitTorrent." *BitTorrent.org Standards Track*. February 13, 2008 (last updated March 25, 2015). http://www.bittorrent.org/beps/bep_0015.html.

[84] The Invisible Internet Project. 2014. "Bittorrent Over I2P." Last updated May 2014. Accessed November 2016. https://geti2p.net/en/docs/applications/bittorrent.

[85] The Invisible Internet Project. No date given. "NetDB Backend." Accessed November 2016. https://geti2p.net/en/get-involved/bounties/netdb.

[86] The Invisible Internet Project. 2016. "I2P Compared to Tor." Last updated November 2016. Accessed November 2016. https://geti2p.net/en/comparison/tor.

[87] DJ Khaled. No date given. Quoted in *TheyDontWantYouTo.Win*. Accessed November 2016. http://www.theydontwantyouto.win/.

[88] Hazel, Greg and Norberg, Arvid. 2008. BEP 9 "Extension for Peers to Send Metadata Files." *BitTorrent.org Standards Track.* January 31, 2008 (last modified May 16, 2016). http://bittorrent.org/beps/bep_0009.html.

[89] Zetter, Kim. 2016. "Internet Archive Successfully Fends Off Secret FBI Order." *The Intercept*. December 1, 2016. https://theintercept.com/2016/12/01/internet-archive-fends-off-secret-fbi-order-in-latest-victory-against-nsls/.

[90] Farivar, Cyrus. 2016. "Yahoo's CISO resigned in 2015 over secret e-mail search tool ordered by feds." *Ars Tehnica*. October 4, 2016. http://arstechnica.com/tech-policy/2016/10/report-fbi-andor-nsa-ordered-yahoo-to-build-secret-e-mail-search-tool/.

[91] Perlroth, Nicole and Goel, Vindu. 2016. "Defending Against Hackers Took a Back Seat at Yahoo, Insiders Say." *New York Times*. September 28, 2016. http://www.nytimes.com/2016/09/29/technology/yahoo-data-breach-hacking.html?_r=0.

[92] The Invisible Internet Project. 2016. "The Network Database." Last updated February 2016. Accessed November 2016. https://geti2p.net/en/docs/how/network-database.