Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs

Scott Wolchok†[1], Owen S. Hofmann†[2], Nadia Heninger[3], Edward W. Felten[3],
J. Alex Halderman[1], Christopher J. Rossbach[2], Brent Waters*[2], and Emmett Witchel[2]

[1]The University of Michigan
{*swolchok,jhalderm*}*@eecs.umich.edu*

[2]The University of Texas at Austin
{*osh,rossbach,bwaters,witchel*}*@cs.utexas.edu*

[3]Princeton University
{*nadiah,felten*}*@cs.princeton.edu*

September 18, 2009

Abstract

Researchers at the University of Washington recently proposed Vanish [19], a system for creating messages that automatically "self-destruct" after a period of time. Vanish works by encrypting each message with a random key and storing shares of the key in a large, public distributed hash table (DHT). Normally, DHTs expunge data older than a certain age. After they expire, the key is permanently lost, and the encrypted data is permanently unreadable. Vanish is an interesting approach to an important privacy problem, but, in its current form, it is insecure. In this paper, we defeat the deployed Vanish implementation, explain how the original paper's security analysis is flawed, and draw lessons for future system designs.

We present two Sybil attacks against the current Vanish implementation, which stores its encryption keys in the million-node Vuze BitTorrent DHT. These attacks work by continuously crawling the DHT and saving each stored value before it ages out. They can efficiently recover keys for more than 99% of Vanish messages. We show that the dominant cost of these attacks is network data transfer, not memory usage as the Vanish authors expected, and that the total cost is two orders of magnitude less than they estimated. While we consider potential defenses, we conclude that public DHTs like Vuze probably cannot provide strong security for Vanish.

Update – September 28, 2009 After we shared these findings with the Vanish team, they released a software update that attempts to defend against our attacks [20] and a report detailing potential countermeasures [18]. We respond to these developments in the update section at the end of this paper.

1 Introduction

As storage capacities increase and applications move into the cloud, controlling the lifetime of sensitive data is becoming increasingly difficult. Even if users cleanse their local files, copies may be retained long into the future by email providers, backup systems, and other services, and these may be targets of theft

*Supported by NSF CNS-0716199 and CNS-0915361, Air Force Office of Scientific Research (AFO SR) under the MURI award for "Collaborative policies and assured information sharing" (Project PRESIDIO).

† Both authors contributed equally.

or subpoena. Geambasu, Kohno, Levy, and Levy proposed the Vanish system [19] to address this problem. Vanish encapsulates data objects so that they "self-destruct" after a specified time, becoming permanently unreadable. It encrypts the data using a randomly generated key and then uses Shamir secret sharing [36] to break the key into n shares where k of them are needed to reconstruct the key. Vanish stores these shares in random indices in a large, pre-existing distributed hash table (DHT), a kind of peer-to-peer network that holds key-value pairs. The encrypted data object together with the list of random indices comprise a "Vanishing Data Object" (VDO).

DHTs have a property that seemingly makes them ideal for this application: they make room for new data by discarding older data after a set time. The DHT policy to age out data is what makes Vanish data vanish. A user in possession of a VDO can retrieve the plaintext prior to the expiration time T by simply reading the secret shares from at least k indices in the DHT and reconstructing the decryption key. When the expiration time passes, the DHT will expunge the stored shares, and, the Vanish authors assert, the information needed to reconstruct the key will be permanently lost. The Vanish team released an implementation based on the million-node Vuze DHT, which is used mainly for BitTorrent tracking.

Vanish is an intriguing approach to an important problem[1]; unfortunately, in its present form, it is insecure. In this paper, we show that data stored using the deployed Vanish system can be recovered long after it is supposed to have been destroyed, that such attacks can be carried out inexpensively, and that alternative approaches to building Vanish are unlikely to be much safer. We also examine what was wrong with the Vanish paper's security analysis, which anticipated attacks like ours but concluded that they would be prohibitively expensive, and draw lessons for the design of future systems.

Attacks Vanish's security depends on the assumption that an attacker cannot efficiently extract VDO key shares from the DHT before they expire. Suppose an adversary could continuously crawl the DHT and record a copy of everything that gets stored. Later, if he wished to decrypt a Vanish message, he could simply look up the key shares in his logs. Such an attacker might even run a commercial service, offering to provide the keys for any Vanish message for a fee. Thus, a method of efficiently crawling the DHT enables a major attack against Vanish.

The authors of this paper represent two groups of researchers who simultaneously and independently developed such attacks. (The Michigan and Princeton authors are one group; the Texas authors are the other.) After discovering that we had separately achieved similar results, we decided to write a joint paper. We show that we can extract the contents of the Vuze DHT using low-cost Sybil attacks.[2] Vuze DHT clients periodically replicate the data they store to other peers that are close-by in the system's ID space. In our attacks, we participate in the network with a large number of identities and record the data that is replicated to them.

The size of the Vuze DHT makes Sybil attacks challenging, as there are typically around a million peers. We investigated two strategies for making our attacks more efficient, which we call "hopping" and "cozying." Hopping significantly reduced the cost of the attacks—as well as the load they placed on the DHT—while enabling us to record enough of the DHT's contents to enable near-complete VDO recovery. Additional optimizations in our implementations brought further savings in CPU, memory, storage, and bandwidth consumption.

The Vanish authors explicitly considered Sybil attacks against the DHT and estimated the cost to be around $860K per year. In contrast, our most efficient attack would cost only $5900 per year to operate at a level that would recover 99% of VDOs. (Both figures are based on Amazon EC2 pricing [3].) Our

[1] Vanish was awarded Outstanding Student Paper at USENIX Security 2009.

[2] In a Sybil attack [15], a single entity assumes many identities within a peer-to-peer network in order to gain control over a large fraction of the system.

optimizations drastically reduce the cost of crawling the Vuze DHT, illustrating that it is possible to defeat Vanish without extraordinary financial resources.

Analysis One of the goals of security research is to learn how to build secure systems. It is instructive to study why systems fail, particularly when those systems set out to provide well-defined security properties. To this end, we ask why Vanish failed and draw a number of lessons for future systems.

We begin by examining the security analysis contained in the Vanish paper and pointing out several shortcomings. The authors dramatically overestimated the cost of running Sybils, failed to anticipate the efficiency gains from optimized attack strategies, and did not notice that recovery scales with coverage in a way that favors attackers. These errors caused the Vanish analysis to overestimate the cost of a successful attack by more than two orders of magnitude.

The Vanish paper also overlooked previous work that might have made Sybil attacks seem like a more credible threat. Similar approaches for crawling DHTs have been applied to other Kademlia-family networks in several measurement studies [28, 37]. The Vanish authors apparently were unaware of these studies, which we survey in Section 7.

There are a number of possible defenses that could be applied to future versions of Vanish and Vuze, including reducing replication, imposing further restrictions on node IDs, and employing client puzzles. Changes like these might make Sybil attacks more expensive, but probably not by a large enough factor to provide strong security for Vanish. Another approach would be to switch from a public DHT, where anyone can serve as a peer, to a privately run system like OpenDHT [35]. Though this would remove the threat from Sybils, the private system would essentially act as a trusted third party, which Vanish was designed to avoid.

Vanish's weaknesses are not only of academic concern. Since the Vanish prototype was released to the public amidst widespread media coverage [25], users may already be treating it as a production system and entrusting it with sensitive data. One might believe that this cannot make matters any worse—the system provides an additional layer of protection, which, if compromised, is no worse than what the user had in place originally. This argument assumes that users' behavior will not be affected by the perceived benefits that Vanish delivers, which seems to us unlikely. For example, a user might not feel compelled to delete an email if he believes that the Vanish system has expired the contents. Or a user might be less scrupulous in the contents she adds, if she thinks the contents will vanish in a few hours. "Why bother to prune my own data," the user may ask, "if Vanish is doing it for me?"

Vanish, as it is deployed today, does not meet its goal of providing self-destructing data. While Vanish's general approach may yet turn out to be viable, our results suggest that implementing Vanish securely remains an open problem.

Organization The remainder of this paper is structured as follows. Section 2 provides further background about Vanish and the Vuze DHT. Section 3 describes our attacks and Section 4 evaluates their performance. We analyze problems with the Vanish design and the Vanish paper's security analysis in Section 5. We consider possible defenses in Section 6, survey related work in Section 7, and conclude in Section 8.

2 Background

This section provides a technical overview of the Vanish system and the Vuze DHT. We refer the reader to the Vanish paper [19] and the Vuze web site [1] for additional details.

2.1 Vanish

The television show *Mission: Impossible* famously began with Jim Phelps receiving instructions from a recording that subsequently self-destructs. Messages that self-destruct at a predetermined time would be useful in a digital context too—especially where privacy is important—though a self-destruction feature is challenging to provide.

For example, the sender of an email might want the contents discarded after the message is read. Even in circumstances where the receiver is agreeable to the sender's wishes, she may be unmotivated to put extra effort toward seeing that those wishes are carried out. We call this model of user behavior *trustworthy, but lazy*.

It has long been known that data retention can be managed by encrypting data and then controlling the lifetime of the decryption key, such as by scheduling the automatic deletion of the key after a predetermined interval [7]. Geambasu, Kohno, Levy, and Levy extend this idea in the Vanish system [19] by employing an intriguing new technique to expire keys. Vanish stores keys in a distributed hash table (DHT); DHTs erase old data after a period of time to make room for new stores, and Vanish exploits this property to ensure that its keys will expire at a predictable time with no intervention from the user.

Vanish uses two principal mechanisms. The first is an *encapsulate* algorithm that takes a data object D as input and produces a Vanishing Data Object (VDO) as output. The second is a *decapsulate* algorithm that accepts as input a VDO and reproduces the original data, with the caveat that decapsulation must be done within a certain time T of the VDO's creation.

Encapsulate The encapsulation algorithm takes as input data D. The algorithm generates a random secret key K, and then encrypts the data D under the key K to yield ciphertext C. Next, the algorithm uses Shamir secret sharing [36] to split the key K into n shares $K_1, \ldots, K_n$ where k shares are required to reconstruct the key. Shamir secret sharing guarantees that k shares of $K_1, \ldots, K_n$ are sufficient to reconstruct K, but no information about the original key K can be recovered with fewer than k shares.

Next, the algorithm chooses a random "access key" L, which is used as a seed to a pseudorandom number generator (PRNG). The algorithm runs the PRNG to derive n indices $I_i, \ldots, I_n$. For $j = 1, \ldots, n$ it stores key share K_j at index I_j in the DHT. Finally, the VDO V is outputted as the tuple $V = (C, L)$.

Decapsulate The decapsulation algorithm accepts a VDO $V = (C, L)$ as input. The algorithm seeds the PRNG with the access key L to retrieve n indices $I_1, \ldots, I_n$. It then retrieves the data values from the DHT at these indices. If fewer than k values are retrieved, the algorithm outputs failure. Otherwise, it uses Shamir secret sharing on k shares to reconstruct a key K'. Finally, it attempts decryption of C using K'. The algorithm outputs a failure if the decryption is not successful; otherwise, it returns D, the result of the decryption.

Security model and assumptions The goal of Vanish is to provide a type of forward security, where past objects are secure if the VDO is compromised after its expiration time. This is somewhat similar to forward secure signatures [6, 22] and forward secure encryption [9]. However, in these systems, a user's machine is responsible for evolving (updating) a private key. In Vanish, the goal is to achieve security without requiring active deletion of the VDO from the user's machine. Instead, the system relies on the DHT data retention policy to expire the shares of the key used to encrypt the VDO.

Consider a user that creates a VDO V with expiration time T. If Vanish is secure then any attacker obtaining the VDO at time $T + t$, $t > 0$ will not be able to reconstruct the data. Providing this guarantee of security requires *at least* two assumptions about the attacker. (We note that if we require *correctness* we also must assume availability of the DHT before the timeout.) These assumptions are:

Limited Network View The attacker must not be able to view the user's traffic to the DHT. Otherwise, the attacker could simply sniff and record the shares as they are stored.

Limited View of DHT The attacker must not be able to read more than a small fraction of the data stored on the DHT. Otherwise, the attacker could build an archive of the DHT and later look up the key shares.

Our work focuses on violating the second assumption.

Deployed implementation Vanish version 0.1 was released in August 2009. It consists of two components: a core system, which provides the encapsulation and decapsulation functions, and a Firefox web browser extension. The Firefox extension allows the user to right-click on a selected area of text to encapsulate it into a Vanishing Data Object. A user can then right-click on the VDO to have the extension retrieve the original text.

By default, the Firefox extension breaks the encryption key into $n = 10$ shares with a recovery threshold of $k = 7$. However, the Vanish paper recommends a slower but more secure setting of $n = 50$ and $k = 45$. The Vuze DHT expires stored data after $T = 8$ hours. Users can extend this time by periodically reposting their key shares.

2.2 Vuze DHT

The current Vanish implementation stores keys in the Vuze DHT, which is used by the Vuze BitTorrent client (also known as Azureus) for decentralized torrent tracking. Vuze estimates that the DHT contains over a million nodes, though there is significant diurnal variation. The DHT is based on a modified Kademlia [27] implementation and functions similarly to many other DHTs. Nodes in the network and keys in the hash table are assigned 160-bit identifiers (IDs). Each DHT node stores those keys which are closest to it in the ID space, as determined by the XOR distance metric. Each Vuze client maintains a routing table that categorizes peers into a number of "k-buckets" by their distance from its own ID, where k is a parameter of the Kademlia design and is set to 20 in Vuze.

All operations are performed using simple RPC commands that are sent directly to the remote peer in a single UDP packet. The primary Vuze RPCs are PING, which checks node liveness and announces the sending node, STORE, which stores a set of key value pairs at the receiving node, FIND-NODE, which requests the 20 closest contacts to a given ID that the receiver has in its routing table, and FIND-VALUE, which functions like FIND-NODE except that the receiver instead returns the values stored at the requested key.

The fundamental Kademlia operation is node *lookup*, which finds the 20 closest nodes to a given ID. To begin a lookup, a node sends FIND-NODE requests to the 20 closest nodes to the ID that it currently has in its routing table. Each peer returns a list of the peers it knows that are closest to the desired ID. The requesting node contacts those peers, reaching successively closer peers until it finds those responsible for storing the desired ID. A lookup terminates when the closest known peer that has not yet been contacted is farther from the desired ID than the farthest of the closest 20 responding peers.

To retrieve or store a value, the requesting node hashes the key to obtain its ID and performs a lookup for the ID to determine the 20 closest peers to the key. It then directly contacts those peers with a request to return or store the associated value. Stores and retrievals are performed on the 20 nodes closest to the desired ID.

A node joins the Vuze DHT by contacting a known peer and initiating a lookup for its own ID. It uses this lookup to build its list of peers and eventually finds the nodes closest to its ID. When a node is contacted by a new peer with an ID among the 20 closest to its own, it replicates all of its stored keys to that node. To minimize unnecessary network traffic, a node only replicates those keys to which it is the closest. To deal with unreliable nodes, peers also replicate the data periodically to the 20 closest nodes to the key's ID.

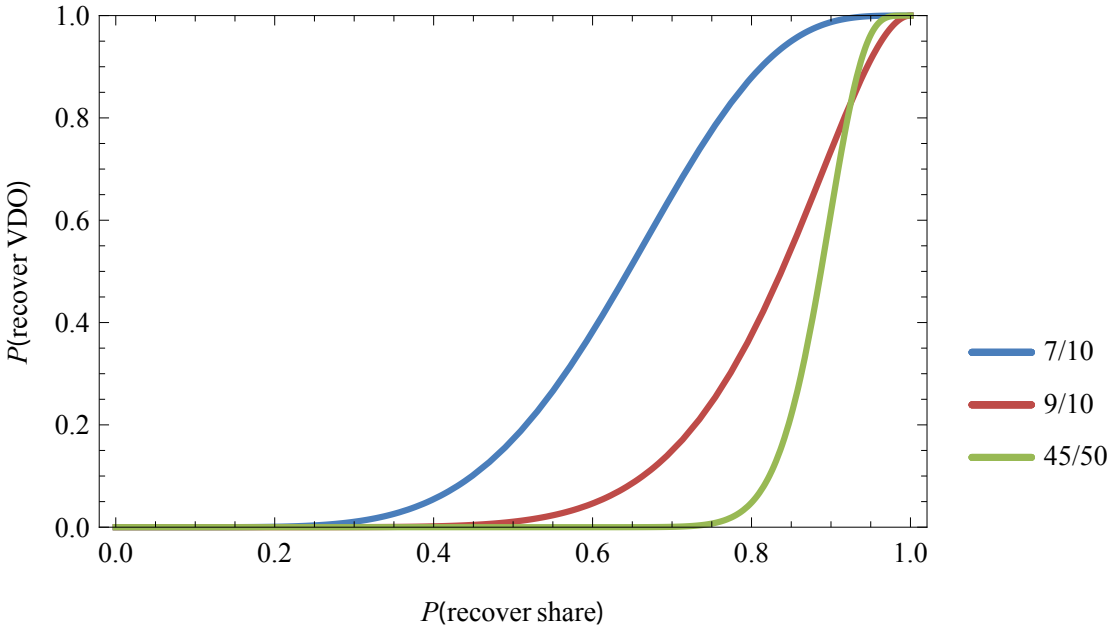

Figure 1: **VDO recovery vs. key share recovery.** The attacker's chances of successfully decrypting a VDO improve rapidly with the probability of recovering any given key share from the DHT. Here we estimate the VDO recovery probability for three pairs of secret sharing parameters k/n.

The Vuze DHT employs a rudimentary anti-Sybil mechanism: node IDs are forced to equal the hash of the node's IP address and port number. This design accommodates the common case of multiple users communicating via a single NAT device by allowing the same IP to join the network at different locations using different port numbers.

3 Attacking Vanish

One way to attack Vanish is with a large Sybil attack against the underlying Vuze DHT. Vuze nodes replicate the data they store to up to 20 neighboring nodes, so a straightforward attack would be to have many Sybils participate in the network and wait for replication to occur.

Figure 1 shows the probability of recovering a VDO given the probability of recovering any individual key share. We model the probability of recovering a key share as a binomial random variable and write the probability of recovering the k/n VDO as

$$\Pr[\text{recover VDO}] = \sum_{i=n-k}^{n} \binom{n}{i} p^i (1-p)^{n-i}$$

where p is the probability of recovering an individual share. The Vanish authors approximated this as a linear function, but that is a poor model of the actual behavior. The probability of recovery exhibits a threshold phenomenon that works to the attacker's advantage.

This figure shows that, to achieve high VDO recovery, the attacker needs to have at least an 80% chance of learning each stored share. Our experiments suggest that this would require more than 60,000 Sybils. Although Vuze allows each IP address the attacker owns to participate with up to 65,535 node IDs (one for each UDP port), the attacker may not have sufficient computing resources to maintain so many Sybils concurrently, and the necessary bandwidth might also be prohibitively high.

The attacker can do much better by exploiting the fact that he does not need continuous control over such a large fraction of the network. Rather, he need only observe each stored value briefly, at some point during its lifetime. Two properties of Vuze's replication strategy make this easy. First, Vuze replicates values to new clients as soon as they join the network. Second, to ensure resiliency as nodes rapidly join and leave, Vuze nodes replicate the data they know to their neighbors at frequent intervals, usually every 30 minutes.

Because of these behaviors, a Sybil node need not participate in the network for very long in order to view the majority of keys available at its position. If the attacker can run m Sybils at a time, they can move, or "hop," through the range of available identities by changing their port or IP address, thus gaining a chance to observe data stored at a different set of locations. Hopping enables the attacker to support $\frac{mt}{T}$ *effective* Sybils during a given period of time t, where T is the duration of each hop.

We found in our experiments that $T = 3$ minutes was sufficient to observe almost all the information stored in the vicinity of each ID. This means that, over the 8 hour VDO lifetime, each Sybil can participate in the network from 160 node IDs with minimal loss in coverage. The hopping strategy vastly increases the efficiency of our attacks.

3.1 Simple Hopping Implementation (Unvanish)

We constructed two implementations to experiment with the hopping attack strategy. The first, *Unvanish*, demonstrates the simplicity of constructing a Sybil attack against Vanish. Unvanish is based on the publicly-available Vuze DHT client code and adds just 268 lines[3] of Java for creating and operating the Sybils. An 82-line Python script instantiates a number of Unvanish processes and controls the nodes' hopping.

Unvanish records keys and values it receives from neighboring nodes upon joining the network. To reduce the cost of storage and transfer to the eventual permanent value archive, Unvanish heuristically discards values that are unlikely to be shares of Vanish encryption keys. A simple heuristic examines the length of stored values. The current Vanish implementation splits encryption keys using Shamir secret sharing over the integers. As a result, the length of shares can vary significantly, depending on key length, number of shares n, and threshold k. Based on the Vanish implementation, we calculate that shares of 128-bit encryption keys for the default setting of $k = 7$, $n = 10$ and the recommended "high-security" setting of $k = 45$, $n = 50$ will be 16 to 51 bytes long, inclusive. Unvanish records all values within that range.

We run Unvanish using the Amazon EC2 system, enabling a realistic assessment of the cost of the attack. We run our Vuze DHT client on 10 of Amazon's "small" EC2 instances, which provide 1.7 GB of physical memory, 160 GB of local storage, and compute power approximately equivalent to a 1.0 GHz Xeon processor. Memory and processor constraints restrict Unvanish to 50 concurrent DHT nodes on each instance. Each DHT node hops to a new node ID after every 150 seconds of operation.

We created an online demonstration of Unvanish that decapsulates VDOs after they have supposedly expired. To minimize the harm to Vanish users, we discard the data we collect from the DHT after one week, though a real attacker could easily keep it indefinitely.

3.2 Advanced Hopping Implementation (ClearView)

In order to investigate how the costs of the attack could be reduced with further optimizations, we developed a second implementation, which we call *ClearView*.

ClearView is a from-scratch reimplementation of the Vuze DHT protocol written in 2036 lines of C. It uses an event-driven design to minimize CPU and memory footprints, and it can run many DHT clients in a single process. It can maintain several thousand concurrent Sybils on a single EC2 instance. On startup, ClearView bootstraps multiple Vuze nodes in parallel, seeding the bootstrap process with a list of peers

[3] All measures of lines of code are generated using David A. Wheeler's 'SLOCCount'.

gleaned ahead of time from a scan of the network in order to avoid overloading the Vuze DHT root node. ClearView then logs the content of each incoming STORE request for later processing.

ClearView reduces the amount of network traffic used in the attack by replying to incoming DHT commands only as necessary to collect stored data. ClearView's Sybils reply to all PING and STORE requests in order to inform other nodes that they are live. Vuze also requires that they respond to a FIND-NODE request before they will receive any STORE requests. Since the principal source of STOREs is replications from nearby nodes, ClearView Sybils reply only to FIND-NODE requests from nodes whose IDs share at least 8 prefix bits with their IDs. ClearView omits the Vuze routing table for efficiency and ease of implementation, so its FIND-NODE replies contain only the contact information of the replying Sybil. ClearView unconditionally ignores FIND-VALUE, KEY-BLOCK, and STATS requests, which are unnecessary for crawling the DHT.

During preliminary experiments, we discovered that Sybils remain in the Vuze routing tables for a significant time after they shut down and that other Vuze peers continue to attempt to contact them. Our hopping strategy causes each Sybil to run for only a short time (3 minutes for our ClearView experiments), so these latent requests amount to substantial unwanted UDP traffic. The problem is compounded by the default behavior of the Linux kernel, which replies to each packet with an ICMP Destination Unreachable message. We found that these ICMP messages constituted a majority of ClearView's outgoing traffic.

We achieved substantial cost savings by simply configuring the Linux firewall to block outgoing ICMP messages. A more advanced implementation might be able to avoid paying for the unwanted inbound traffic as well by using EC2's network firewall API [2] to allow traffic only to ports used by the current set of Sybils.

3.3 An Alternative Strategy

We attempted to further limit routing table integration with a second attack strategy, which we call *cozying*. This approach is divided into two processes: enumeration and crawling. The enumeration process attempts to discover all the peers participating in the DHT by making carefully chosen requests using the DHT's peer discovery mechanism. We pass the resulting list of nodes to the crawling process, which contacts each of the nodes using the closest available node ID and waits to receive replicated data.

Our experiments so far indicate that cozying does not result in a more efficient attack. We noticed significant traffic from nodes we did not contact directly, indicating that we were unable to avoid being integrated into their routing tables. We conjecture that this is because Vuze, unlike other Kademlia-style DHTs, will forward nodes that are not known to be responsive (e.g., received in reply to a FIND-NODE request) to other nodes in reply to FIND-NODE requests, thus propagating them through the routing tables. We plan to investigate this behavior further in future work.

4 Evaluation

This section measures the effectiveness of our two attack implementations and quantifies the costs of running the attacks on Amazon EC2.

4.1 Simple Hopping

We ran Unvanish on 10 "small" EC2 instances for approximately 24 hours. Over a 7.5-hour window during that time, we seeded 104 VDOs into the DHT, using the default security parameters of 7 of 10 shares required for decryption. Each EC2 instance ran 50 concurrent Sybils which hopped every 150 seconds, giving us data from 96,000 node IDs during the 8-hour DHT store lifetime. Out of 1040 key shares, we were able to recover 957, indicating that we achieved about 92% coverage of key-value pairs. We successfully decrypted 100% of the 104 VDOs using the data Unvanish collected.

			Average per effective Sybil		
Effective Sybils	Target values	Targets found	Coverage (s)	Bandwidth in (b_{in}) / out (b_{out})	
320,000	1650	99.4%	0.0030%	15.7 B/s	0.589 B/s
270,000	1700	99.5%	0.0032%	13.3 B/s	0.524 B/s
80,000	1650	91.8%	0.0036%	15.8 B/s	0.625 B/s

Table 1: **ClearView experimental results.** We conducted three trial attacks with our ClearView implementation. Over periods of around 8 hours, Sybils "hopped" to new node IDs every three minutes, yielding 320k, 270k, and 80k effective Sybils. On average, each effective Sybil observed $s = 0.0033\%$ of the DHT. Larger trials had slightly worse per-Sybil coverage, possibly due to network congestion.

Running an EC2 "small" instance costs $0.10 per hour if the instance is created on demand. Amazon also provides reserved instance pricing, which entails an upfront charge followed by a reduced per-hour usage charge. A one-year reservation for 10 instances running full-time would cost $0.56 per hour. During a one day run of Unvanish, our 10 EC2 instances transferred 176 GB of data in and 196 GB out, and the average transfer cost was $2.12 per hour. Extending these figures, the cost for machines and transfer to run Unvanish for a year would be $23,500. By contrast, the original Vanish paper estimates that such an attack would have an annual cost exceeding $860,000.

4.2 Advanced Hopping

To evaluate the effectiveness of ClearView, we ran trial attacks against the Vuze network for around 8 hours at a time. For two hours before the start of each experiment, we used the deployed Vanish client to insert key shares into the DHT from a distant network location. These served as targets for our Sybils.

We ran three trials with different numbers of effective Sybils, as summarized in Table 1. In all three trials, ClearView ran on a number of "small" EC2 instances and used a three minute hop time. The first trial used 10 EC2 instances, each supporting 200 concurrent Sybils over 8 hours, for a total of 320k effective Sybils; there were 1650 target key shares, and we recovered 1640 of them (99.4%). The second trial used 9 EC2 instances, each supporting 200 concurrent Sybils over 7.5 hours, for a total of 270k effective Sybils; there were 1700 target key shares, and we recovered 1692 of them (99.5%). The third trial used 10 EC2 instances, each supporting 50 concurrent Sybils over 8 hours, for a total of 80k effective Sybils; there were 1650 target key shares, and we recovered 1561 of them (91.8%).

These results allow us to estimate the fraction of DHT values that could be observed with different numbers of effective Sybils. We calculate how many of the target values were observed by different size subsets of our effective Sybils. Figure 2 shows the results, based on data from the 270k trial.

As we explain below, we can accurately model the DHT coverage we achieve using only a single parameter, s, the fraction of the DHT observed by each effective Sybil. For our 320k effective Sybil trial, each Sybil observed 0.050 target values on average, yielding average per-Sybil coverage $s = 0.000030$. For our 270k effective Sybil trial, each Sybils observed 0.055 target values on average, yielding average per-Sybil coverage $s = 0.000032$. For the 80k effective Sybil trial, each Sybils observed 0.059 target values on average, yielding average per-Sybil coverage $s = 0.000036$.

An analytic model The shape of the data can be explained using a simple combinatorial model. If we make the approximation that each Sybil sees some fraction of the total network chosen uniformly at random, then we can model the number of unique objects seen by a collection of Sybils as a random process.

In this case, the process is equivalent to the balls-into-bins problem. If each Sybil sees on average c objects from the network and there are m Sybils, then the Sybils together will have collected cm objects.

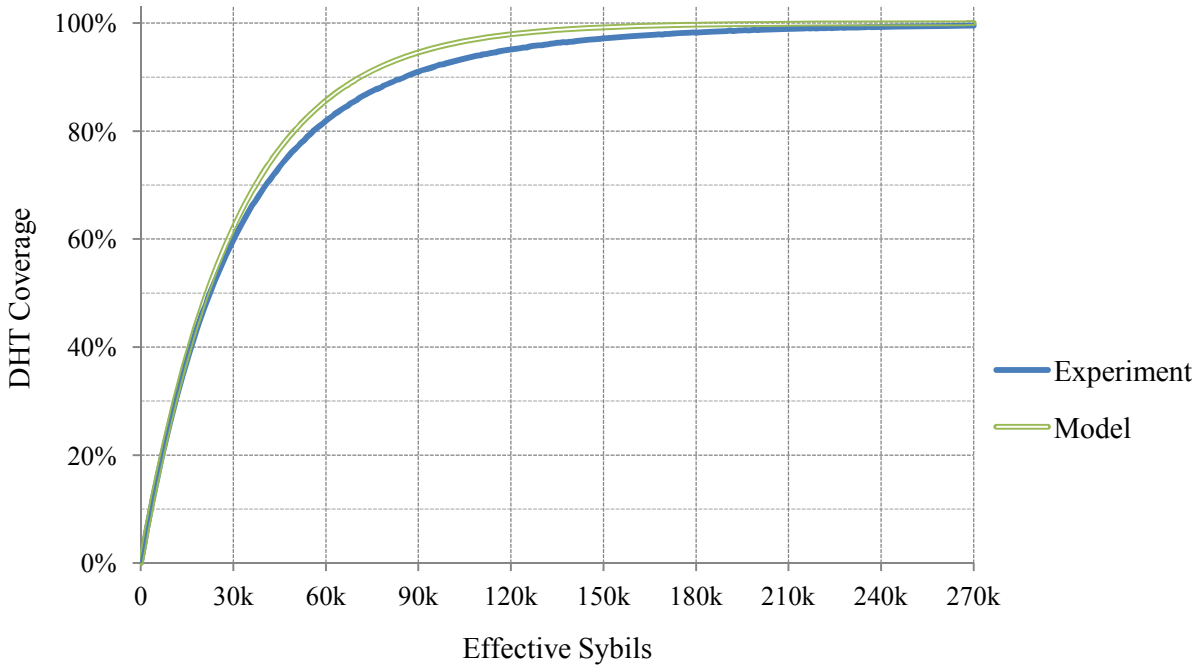

Figure 2: **DHT coverage vs. attack size.** This graph shows the fraction of values stored in the DHT that we would have collected using different numbers of effective Sybils. We derived these results by running ClearView for 8 hours to collect data from 270k node IDs and computing the average coverage for different size subsets. The data closely correspond to a simple model based on the average coverage per effective Sybil, s.

However, once cm is no longer small with respect to the total number of objects N in the network, the Sybils will be very likely to collect repeats of objects and it will be more difficult to discover new objects. This process is equivalent to throwing cm balls into N bins and asking how many bins contain at least one ball; that is, how many objects were seen by at least one Sybil.

The analysis of this problem is standard; see, e.g. [29, Ch. 5.3]. The expected fraction of DHT objects observed by m Sybils is $1 - e^{-cm/N}$. Let s be the fraction of the entire DHT observed by each Sybil, that is, $s = \frac{c}{N}$. We can rewrite the expected fraction of DHT objects observed by an m-Sybil attack in terms of this parameter:

$$\text{E}[\text{DHT objects observed}] = 1 - e^{-ms}.$$

Figure 2 illustrates this model's close correspondence with our experimental results. The model slightly overestimates actual performance due to the simplifying assumption that each Sybil observes an equal number of fragments.

We can use this model of DHT coverage, and the calculation of VDO recovery in terms of key share recovery from Figure 1, to estimate the fractions of VDOs that we would recover with different size attacks. Figure 3 shows the results using data from our 270k effective Sybil trial. For the default Vanish secret sharing parameters of $k = 7$ and $n = 10$, we would need 26k effective Sybils to recover 25% of VDOs, 59k to recover 90%, and 89k to recover 99%. For secret sharing parameters of $k = 9$ and $n = 10$, we would need 48k effective Sybils to recover 25% of VDOs, 115k to recover 90%, and 186k to recover 99%. For the Vanish paper's conservative secret sharing parameters of $k = 45$ and $n = 50$, we would need 70k effective Sybils to recover 25% of VDOs, 107k to recover 90%, and 136k to recover 99%.

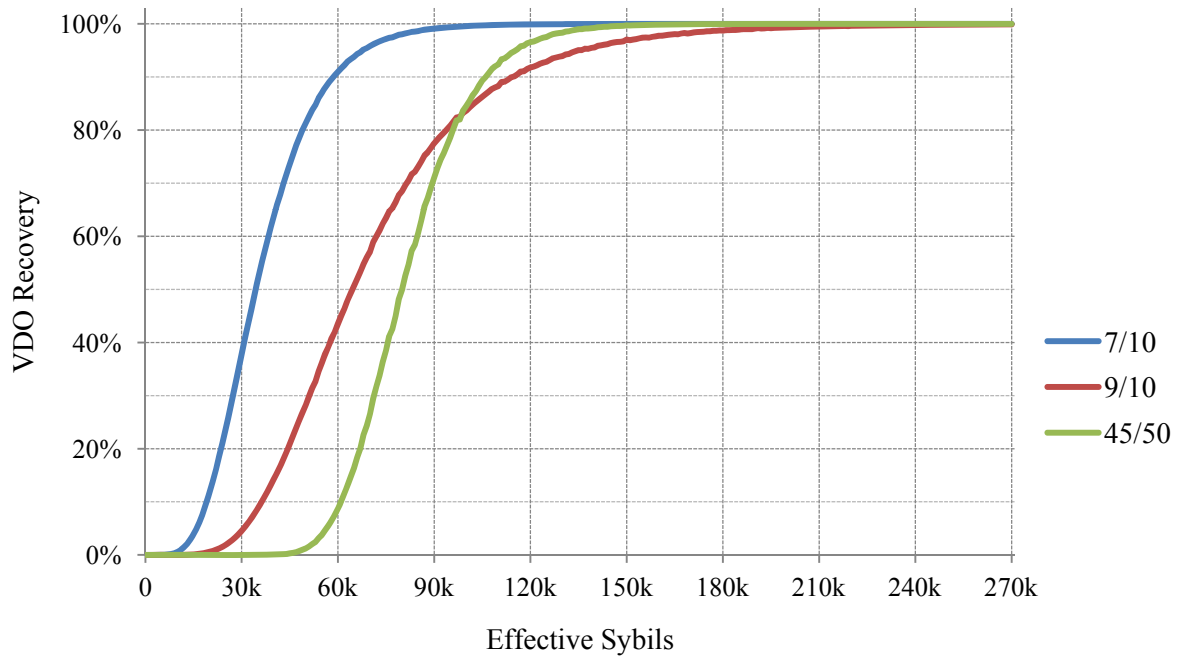

Figure 3: **VDO recovery vs. attack size.** This graph shows the fraction of VDOs that we would be able to decrypt using key shares collected with different numbers of effective Sybils, based on data from the previous figure. The curves correspond to three pairs of secret sharing parameters k/n.

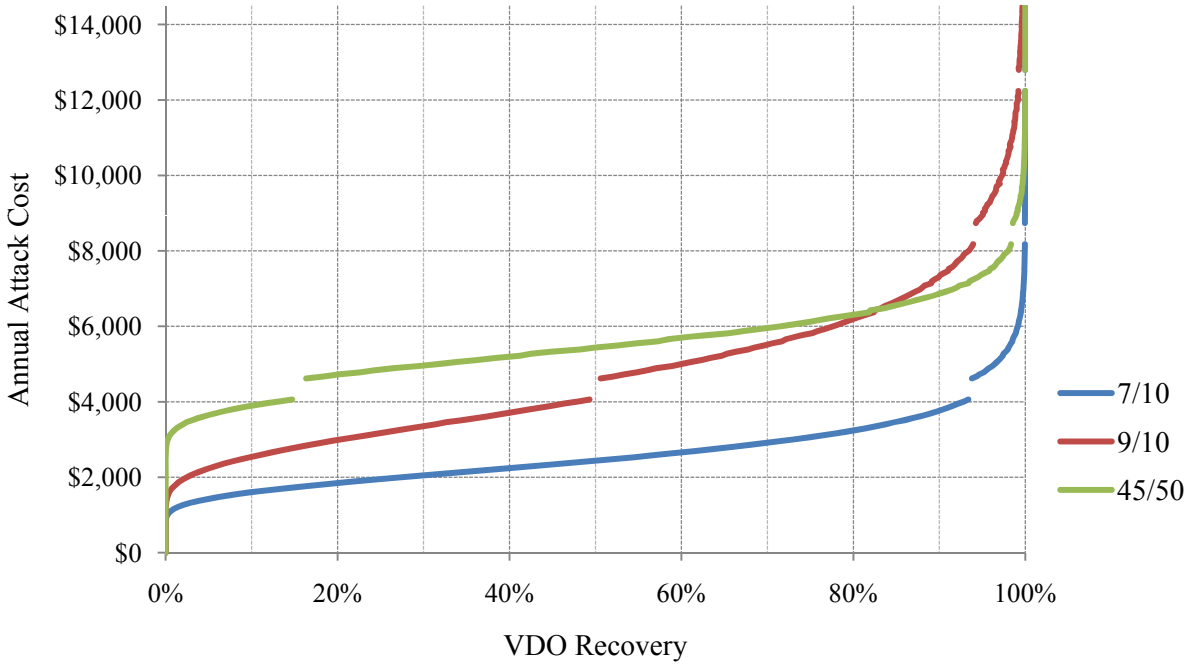

Figure 4: **Attack cost vs. VDO recovery.** Here we plot the costs of using our ClearView attack implementation to recover different fractions of VDOs, based on data from the previous figures. These estimates include the cost of EC2 network transfer as well as the coarse-grained cost of adding machine instances (which results in the discontinuities). The curves correspond to three pairs of secret sharing parameters k/n.

Category	Values (per 8 Sybil-hours)	Bytes (per 8 Sybil-hours)
Peer record	850	4000
Bencoded	330	31,000
Other	220	5000
Total	1400	40,000
Potential Vanish key share	95	1700

Table 2: **Observed stored values.** These figures represent the average number of values and bytes observed by each Sybil during an 8-hour window, without removing duplicates. Currently, Vanish key shares always occupy 16–51 bytes, so an attacker would only need to retain these values.

Machine and network costs Based on these approximations and the costs of running our full experiment, we can extrapolate the costs of longer attacks targeting various fractions of VDOs. The dominant costs are those of machines and network transfer. Each effective Sybil needs a unique IP address and port combination. An EC2 instance can only use a single IP address, so the attacker needs one instance for every 65535 effective Sybils. (ClearView easily support this number on even a "small" instance.) Each instance costs about $500/year with reserved pricing.

Network transfer is the dominant cost for most attack parameter choices. We can estimate the transfer costs based on the average bandwidth used by each Sybil. EC2 prices inbound and outbound transfers differently, so we report averages for both directions. During our 8-hour trial period running 320k effective Sybils, ClearView transferred 145 GB in and 5.5 GB out, for an average bandwidth per effective Sybil of 15.7 B/s in and 0.589 B/s out. Our other trials produced similar results, indicating that the network transfer cost of running an attack for one year would be $52.80–$54.80 per thousand effective Sybils.

Figure 4 estimates the cost of EC2 instances and transfer for year-long attacks aiming to recover various fractions of VDOs. For the default Vanish secret sharing parameters of $k = 7$ and $n = 10$, recovering 25% of VDOs would cost about $1950 per year, 90% would cost about $3750, and 99% would cost about $5900. For secret sharing parameters of $k = 9$ and $n = 10$, recovering 25% of VDOs would cost about $3150 per year, 90% would cost about $7350, and 99% would cost about $11,950. For the Vanish paper's conservative secret sharing parameters of $k = 45$ and $n = 50$, recovering 25% of VDOs would cost about $4850 per year, recovering 90% would cost about $6900, and recovering 99% would cost about $9000.

Storage costs To carry out an ongoing attack, the adversary needs to store the DHT values he collects. We attempted to quantify the costs of this storage by measuring the data collected by our Sybils.

Table 2 shows the average number of values and bytes that were recorded by each Sybil during an 8-hour experimental window. We omit STOREs for empty values, since they represent key deletion in the DHT. Most of the values are related to Vuze's BitTorrent tracking functions. Bencoding [12] is a simple serialization format that is part of the BitTorrent protocol, and peer records are the human-readable values inserted into the DHT to record the presence of peers for DHT-tracked torrents.

To obtain 94% coverage, the attacker would have to run 128,000 effective Sybils over each 8-hour period. Recording every key-value pair for a one-year period would require about 9.5 TB of storage and cost about $1400 (using Amazon S3 pricing [4]).

The attacker can greatly reduce the cost by storing only values that are possible key shares in the current Vanish implementation—those with lengths between 16 and 51 bytes, inclusive. This would amount to less than 510 GB per year and cost under $80. Further savings could be achieved by discarding duplicate stores, though this would require additional post processing.

5 Discussion

Researchers often discover vulnerabilities in systems proposed by other researchers, and the process of learning from these problems has produced fruitful advances. Examples include the development of the Tor anonymous communication system [5, 14, 30–33] and Off-the-Record Messaging [8, 34]. In this section, we discuss problems in the Vanish design and in the Vanish paper's security analysis, and we attempt to draw lessons for future systems.

5.1 Problems with the Vanish Security Analysis

The Vanish paper includes a security analysis that explicitly considers the threat from Sybil attacks against the Vuze DHT. We have shown that the paper's cost estimate—$860,000 for a year-long attack—is two orders of magnitude higher than the cost of our attacks. We now examine how the Vanish paper arrives at this figure, and where the analysis goes wrong.

The Vanish paper estimates the number of Sybils required to compromise 25% of VDOs. Rather than experimenting with the public Vuze network, the authors used their own private deployment consisting of 8000 Vuze DHT clients. For secret sharing parameters of $k = 45$ and $n = 50$, they found that an attacker would need 820 Sybils. Extrapolating this figure, they determined that attacking a network with 1M clients would require 87,000 Sybils.

This approach—simulating a much smaller network and extrapolating—introduces considerable uncertainty and overestimates the actual difficulty. Our experiments show that our attacks require roughly 70,000 effective Sybils to compromise 25% of VDOs under these parameters.

There are two problems with the 87,000 Sybil estimate. The first is that it assumes the Sybils need to run continuously. As we have shown with our hopping attack strategy, each Sybil can run for as little as 3 minutes in each 8 hour period, with little loss in coverage. Thus, only 544 Sybils need to run concurrently to achieve the same effect as 87,000 Sybils running continuously.

The second problem is that estimating the work needed to recover 25% of VDOs provides only a lower bound on the Sybils needed to recover greater fractions. Readers might infer that recovering 90% would take many times more effort. In fact, as we illustrate in Figure 1, the fraction of the DHT that the attacker needs to observe increases only slightly, from 86% to 93%. Our experiments show that, under these parameters, we can compromise 90% of VDOs with 107,000 effective Sybils—only 53% more than are needed to compromise 25% of VDOs.

Based on their estimate that the attacker would need 87,000 Sybils, the Vanish authors calculate that a year-long attack using EC2 would cost at least $860,000 for "processing and Internet traffic alone." This figure is not entirely explained. One clue comes from the design of their experiments, which were conducted in part on EC2. They used the official Vuze DHT client, which is written in Java and has high CPU and memory footprints. They found that the limiting resource was memory—50 nodes would fit in 2 GB of RAM. If we round up the memory available in a small EC2 instance to 2 GB (it is actually 1.7 GB) and assume reserved instance pricing, operating 87,000 nodes would cost $854,000 a year for machine time, which the Vanish authors may have rounded up to arrive at their figure.

One problem with this estimate is that an actual attacker can write a much more efficient Vuze client, as we show with our ClearView implementation. With our optimized client, we can support thousands of concurrent Sybils in a single small EC2 instance. Network transfer is the limiting cost, not memory or CPU. Another problem is that the amount of traffic used for the attack is very difficult to estimate without participating in the real network. It depends on empirical factors like the amount of routing traffic and the rate of stores. This adds further uncertainty to the Vanish paper's cost estimate.

Recovering 25% of VDOs with our most efficient attack would cost less than $5000 per year—more than

100 times less than the Vanish estimate. This illustrates the value of attempting realistic attacks, even when models and analysis seem to show that a system is secure.

5.2 Problems with the Vanish Design

Vanish relies on a security property that the Vuze DHT was not designed to provide—resistance to crawling. Sometimes repurposing existing systems allows clever solutions to security problems, but we believe it must be done with extreme caution. Computer systems evolve to satisfy the demands of their users and maintainers, so it is risky to rely on them for security properties that are not important to their primary users and maintainers. Even if the current Vuze environment were more favorable to the security of Vanish, the system might evolve in an unfavorable direction.

The worldwide scale of Vuze is essential to the security of Vanish, and the defense needed by Vanish to thwart our Sybil attack is not aligned with the priorities of Vuze's main user base. A large DHT with nodes run by independent people is essential to Vanish's guarantee that secret shares are widely dispersed and safe from collusion. Anything less than a world-wide DHT is unlikely to be secure for Vanish because smaller communities are vulnerable to collusion attacks and social engineering.

Our attacks are made easier, and Vanish suffers, because the Vuze DHT replicates entries twenty times and actively creates replicas periodically (and immediately to newly arrived peers). These features are important for the primary purpose of Vuze, which requires DHT entries to stay in the DHT with high probability despite significant churn in the node population. Here Vuze and Vanish are working at cross purposes. It seems unlikely that Vuze would accept changes that significantly reduce DHT reliability, just for the benefit of Vanish.

At a fundamental level, public DHTs of the sort used by Vuze are not well suited for keeping secrets. Any item can be read by anyone who knows its ID, and the DHT readily accepts membership from diverse, untrusted peers. The jury is still out on whether the kind of distributed storage medium needed to make Vanish secure, useful and efficient can be designed.

5.3 Vanish and the Bounded Retrieval Model

The Bounded Retrieval Model was proposed by Dziembowski [16]. In the Bounded Retrieval Model, an attacker that compromises a machine can only communicate a limited amount of material back to itself. Multiple recent cryptographic systems were proposed in this framework [10, 16]. The security of systems built both in this model and the related Bounded Storage Model [26] depend on the ability to make accurate estimations of the attacker's capabilities. If such an estimation is off by a factor of 10, this will likely be devastating to the security of the overlying system. In contrast, traditional encryption systems provide a super-polynomial gap between the effort required to use the system and that needed to break it. In these systems it is easy to build in a reasonable "safety margin" when choosing a security parameter.

The Vanish authors, in effect, show that their system is secure under the condition that an attacker is bounded in the amount of information he can retrieve from the underlying DHT in a given amount of time. Unfortunately, their estimates of an attacker's capabilities were off by about two orders of magnitude. To the best of our knowledge, Vanish is one of the first systems to actually be implemented that rely on the bounded retrieval model. Its shortcomings suggest that significant caution is due when building such systems in the future.

6 Countermeasures

Future versions of Vanish and Vuze could adopt various countermeasures against crawling attacks. While we discuss several strategies for making these attacks more expensive, it seems difficult to raise the cost enough to provide strong resistance without sacrificing other security goals, usability, or reliability.

Raising Vanish's key recovery threshold The key shares that Vanish stores in the DHT are produced using a k-of-n secret sharing scheme. By default, 7 of 10 shares are required to reconstruct the key. One defense would be for Vanish to use a stronger values for k and n, such as requiring 99 of 100 shares.

This approach is problematic for two reasons. First, since a small fraction of key shares are lost from the DHT before they expire due to churn in the network, raising the recovery threshold will make more Vanish messages self-destruct ahead of schedule. Second, an attacker could react by scraping the DHT more completely.

Switching Vanish to a privately hosted DHT Future implementations of Vanish could switch from the Vuze DHT to a privately hosted DHT. One option would be OpenDHT [24], a DHT system that, until recently, operated on a collection of PlanetLab nodes. OpenDHT allows anyone to store and retrieve values, but, since it is hosted on a closed set of servers, Sybil-based crawling attacks are not possible without insider access.

There are several problems for Vanish security using a smaller-scale DHT, even one with special security features. A DHT with a small user base or a single maintainer is vulnerable to social collusion. In the case of OpenDHT, convincing its single maintainer to add an anti-Vanish feature would compromise the security of Vanish. It is also easier to convince enough participants to subvert the security of a system when the user base is small and drawn from a particular community (e.g., tens of academic users in the case of OpenDHT). Lastly, a privately hosted DHT like OpenDHT would essentially function as a trusted third party, and there are simpler ways to implement Vanish-like behavior in applications where invoking a trusted third party is acceptable.

Adding client puzzles to Vuze Client puzzles have been proposed as a defenses against Sybil attacks [23]. A simple approach would be to require clients to perform an expensive computation tied to the current date and their node ID. For example, if Vuze required a daily computation that took one minute on a small EC2 instance, this would impose a cost of $0.34/year for each Sybil. To obtain 90% VDO recovery, we need 107,000 effective Sybils (for $k = 45$ and $n = 50$), so we would need to devote 74 EC2 instances to solving puzzles. This would raise the cost of our attacks by about $37,000 per year. Though this is a significant increase, it only impacts attackers who actually pay for CPU time—an attacker who controlled even a small botnet could easily perform the puzzle computations. In addition, if the puzzles were predictable, then an attacker might use precomputation to solve several puzzles for a certain time period. While the attacker might not be able to sustain this attack, all VDO created during this time period would be vulnerable.

Detecting attackers Another possible defense is to try to detect attackers and selectively block or penalize their interactions with Vuze. One approach would be to monitor peers for deviations from the Vuze protocol that distinguish them from legitimate clients. This is currently easy to do for our ClearView software, which omits certain functionality for ease of implementation, but attackers might try to avoid detection by responding to requests more faithfully. A second approach would be to monitor IP addresses that host an unusual number of Vuze clients. Instrumenting the Vuze bootstrap node or scanning the routing tables maintained by peers in the network would detect such IPs.

We experimented with the latter approach by building a tool called *Peruze*, a routing table scanner for the Vuze DHT. Peruze enumerates the nodes in the network by breadth-first search through the node ID space

and iteratively dumping the buckets in each node's routing table. To extract the contacts of bucket i on node N, Peruze sends N a FIND-NODE request for N's ID with the ith bit from the left complemented. To avoid overloading the network, it only sends one FIND-NODE at a time to a given node. Peruze terminates the scan when it has sent packets at a rate less than 10% of its maximum for 30 seconds. Peruze consumes very little bandwidth because it avoids being incorporated into other nodes' routing tables and ignores all messages other than replies to its FIND-NODE requests. Complete scans of the network take under an hour.

We found that, while the majority of IP addresses were associated with only a single node, our Sybil machines were each associated with thousands. Even after we terminated the attacks, these traces persisted in the routing tables for several hours.

Peruze also detected other EC2 nodes not controlled by us with an unusual number of entries in the routing tables, as well as a set of 10 machines at the University of Washington that the Vanish authors confirmed are used to support the online Vanish demonstration. This suggests that techniques like Peruze can be used to detect monitoring and experimentation on the Vuze network. Whether effective countermeasures can be taken once attacks are detected remains an open question.

Restricting node IDs in Vuze The Vuze DHT implements a basic Sybil defense by restricting how node IDs are assigned. The node ID is a function of the client's IP address and port, yielding $2^{16} - 1$ node IDs per IP. This allows us to support 131,070 effective Sybils with only 2 IP addresses.

Future versions of the protocol may restrict the number of IDs attainable from a single IP to 1999. (Contrary to statements in the Vanish paper, this defense is not currently active.) If this change were deployed, we would need 66 IP addresses to obtain the same number of effective Sybils.

Even stronger restrictions might be possible. In our Vuze routing table measurements, we found instances where single IPs were supporting 30 or more clients; these appeared to be legitimate ISPs using NAT. Vuze could limit each IP to, say, 64 identities. We would then require 2048 IP addresses (four class-C blocks) to do the work of 2 addresses today.

These defenses might not have a significant effect on the cost of our attack, since they do not actually increase the required number of machines or the amount of traffic generated. While it is currently difficult to use more than one IP address on an EC2 instance, other ISPs may be willing to rent out unused address blocks at low cost. Alternatively, an attacker who controlled a small botnet would have ready access to addresses.

Securely deploying such defenses on Vuze would take time, due to the need to maintain backwards compatibility with older clients. Until the defense is enforced, an attacker could use an older version of the protocol to circumvent the defense. The most recent Vuze source maintains backwards compatibility to version 3.1.1.1, released more than a year ago. Breaking such backwards compatibility would be a significant, undesirable change to the usability of Vuze.

Regardless of what other countermeasures might be proposed, security claims for Vanish, like all systems, should be treated with respectful skepticism. Discussion of countermeasures is useful, but, as usual, it is prudent to treat a system as insecure until its security is firmly established.

7 Related Work

The original Vanish paper overlooked previous work that could have served as an important indicator about the threat of Sybil attacks for monitoring DHTs. Approaches for crawling DHTs that are similar to our attacks have been applied to other Kademlia-family networks. Other work has attempted to enumerate the nodes in the network, both for Vuze and for other Kademlia-style systems. We survey these works here.

DHT enumeration and monitoring Stutzbach and Rejaie developed *Cruiser* [39] in order to enumerate nodes on the Gnutella network. Cruiser uses a master-slave architecture, employing multiple desktop PCs to enumerate the network in parallel. It also has an adaptive flow-control algorithm based on CPU load in order to maximize the number of parallel connections. Stutzbach and Rejaie extended Cruiser [40,41] to enumerate nodes in the Kad DHT, which is used by the popular eMule program and is also based on Kademlia. Citing very long run times, they chose to enumerate subnets based on fixing prefix bits of the node IDs. In addition, their *kFetch* [40] tool efficiently downloads Kad peers' entire routing tables. Stutzbach and Rejaie's work focuses on accuracy, not cost-effectiveness. In a similar vein, Steiner et al. created *Blizzard* [38], a fast Kad enumerator. Unlike Cruiser, Blizzard uses one PC, keeps all state in memory, and can enumerate all nodes in the Kad network in 8 minutes.

Falkner et al. [17] previously measured the responsiveness, consistency, and performance of the Vuze DHT. Their work used tens of instances of a version of the official Vuze (then Azureus) client modified to collect statistics. Crosby and Wallach [13] also profiled the Vuze DHT as well as the "Mainline" DHT shared by many other BitTorrent clients. They too used instrumented clients to obtain their measurements, but only ran 11 concurrent clients. Several of the parameters of the Vuze DHT have since been modified; in particular, the message timeout and the number of concurrent messages have both been halved.

Our node enumerator, Peruze, omits many optimizations used in Cruiser and Blizzard for the sake of expedient implementation. In particular, its flow control algorithm is naive and it is at least 5 times slower than Blizzard. Unlike previous work, Peruze terminates when the rate of outgoing traffic drops significantly, which indicates diminishing returns from further scanning, rather than continuing in order to obtain more complete measurements.

Crawling DHTs *Mistral* [37] is a crawler for Kad. It carries out a Sybil attack on Kad by efficiently implementing many Sybils on one machine. Instead of using the standard Kad bootstrap, Mistral uses Blizzard to discover peers and then contacts them directly. Mistral does not attempt to discover content through replication; instead, the Sybil nodes always redirect routing traffic to other Sybils to maximize the amount of traffic captured. Mistral is only capable of spying on an 8-bit prefix (i.e., $\frac{1}{256}$) of Kad at a time.

Montra [28] is a crawler for Kad that improves on Mistral by minimizing disruption to the DHT. Like Mistral, it uses a crawler (Cruiser) to discover nodes in the network. Unlike Mistral, Montra discovers content through the replication mechanism by targeting peers individually. At the time Montra was developed, Kad peers were permitted to choose their IDs arbitrarily, so each of Montra's *minimally visible monitors* can set its ID to differ from that of its target peer only in the least significant bit, ensuring replication of stored data. Montra minimizes load and disruption by responding only to each monitor's target peer. However, Montra is not compatible with the new Sybil attack protections in Kad described in [11] that prevent nodes from choosing their Kad IDs arbitrarily.

Goel et al. [21] point out that choosing node IDs in DHTs by hashing the IP address and a salt facilitates the Sybil attack because attackers can choose the salt freely. This observation extends trivially to ports in the Vuze DHT.

Our advanced hopping attack system, ClearView, is distinct from both Mistral and Montra, although it is broadly similar in that it is an efficient implementation allowing many Sybils to run on a single machine. Like Montra, it learns about stored values through replication, but its Sybils join the network through the standard bootstrap procedure. ClearView attempts to restrict its visibility by failing to reply to certain messages, but does not go to the same extent as Montra. Also unlike these previous works, our attacks target the Vuze network and are optimized for attacking the Vanish system.

8 Conclusion

The security guarantees that Vanish sets out to provide would be extremely useful, but, unfortunately, the system in its current form does not provide them in practice. As we have shown, efficient Sybil attacks can recover the keys to almost all Vanish data objects at low cost. Changes to the Vanish implementation and the underlying Vuze DHT might make Sybil attacks somewhat more expensive, but it seems doubtful that such defenses would make the system sufficiently secure. While we would like to see the Vanish system succeed, we are skeptical that it can be implemented securely.

Acknowledgments

We thank the Vanish authors for valuable and constructive discussions. We are also grateful to Dan Wallach for providing machines for our early experiments. We thank Adam Klivans for emotional energy and gravitas.

Update – September 28, 2009

After we shared these findings with the Vanish team, they released a new version of the software [20] and a report [18] detailing potential new defenses. They propose two main countermeasures. The first is to store Vanish keys on both the Vuze DHT and OpenDHT so that data from both DHTs would be needed to recover the key. They implemented this defense in the new Vanish software release, version 0.2. The second defense is to modify the Vuze DHT to disable the push-on-join behavior and use less aggressive data replication. (To minimize the impact on Vuze, they suggest that these changes could be selectively enabled for Vanish data.) The Vanish authors are working with the makers of Vuze to implement this defense.

We are still evaluating these proposals, but we offer our initial perspectives here. Using both OpenDHT and Vuze might raise the bar for an attacker, but at best it can provide the maximum security derived from either system—if both DHTs are insecure, then the hybrid will also be insecure. OpenDHT is controlled by a single maintainer, who essentially functions as a trusted third party in this arrangement. It is also susceptible to attacks on roughly two hundred PlanetLab nodes on which it runs, most of which are housed low-security research facilities. The new Vanish technical report acknowledged OpenDHT's limitations: "For Vanish, OpenDHT seemed a poor fit for a number of reasons" [18, Section 3.2]. Using both Vuze and OpenDHT seems unlikely to be a much better fit.

Changing the behavior of the Vuze DHT might make our attacks more expensive, but it is difficult to gauge *how much* more expensive until these changes are deployed. Understanding their effects on Vanish (and on overall DHT performance) will require significant investigation, and it is possible that entirely new attacks will emerge. While Vuze's willingness to adopt changes for the benefit of Vanish is laudable, it is also a reminder that the Vuze DHT is effectively under the control of a single party, and that future changes could unintentionally or maliciously degrade Vanish's security.

The first iteration of Vanish was broken in a relatively short time. The proposed new defenses are interesting and merit further investigation, but, for the time being, Vanish's security should be viewed with skepticism. Whether DHTs are the best choice for key-share storage remains an open question.

References

[1] *Vuze web site*. http://www.vuze.com.

[2] Amazon. EC2 user guide: Using network security. http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/using-network-security.html.

[3] Amazon. Pricing of EC2. http://aws.amazon.com/ec2/#pricing.

[4] Amazon. Pricing of S3. http://aws.amazon.com/s3/#pricing.

[5] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker. Low-resource routing attacks against Tor. In *WPES*, pages 11–20, 2007.

[6] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In *CRYPTO*, pages 431–448, 1999.

[7] D. Boneh and R. J. Lipton. A revocable backup system. In *Proc. USENIX Security Conference*, pages 91–96, 1996.

[8] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, pages 77–84, 2004.

[9] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, pages 255–271, 2003.

[10] D. Cash, Y. Z. Ding, Y. Dodis, W. Lee, R. J. Lipton, and S. Walfish. Intrusion-resilient key exchange in the bounded retrieval model. In *TCC*, pages 479–498, 2007.

[11] T. Cholez, I. Chrisment, and O. Festor. Evaluation of Sybil attack protection schemes in KAD. In *AIMS '09: Proceedings of the 3rd International Conference on Autonomous Infrastructure, Management and Security*, pages 70–82, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] B. Cohen et al. BitTorrent protocol specification v1.0.

[13] S. A. Crosby and D. S. Wallach. An analysis of BitTorrent's two Kademlia-based DHTs. Technical Report TR-07-04, Department of Computer Science, Rice University, June 2007.

[14] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320, 2004.

[15] J. R. Douceur. The Sybil attack. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[16] S. Dziembowski. Intrusion-resilience via the bounded-storage model. In *TCC*, pages 207–224, 2006.

[17] J. Falkner, M. Piatek, J. P. John, A. Krishnamurthy, and T. Anderson. Profiling a million user DHT. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 129–134, New York, NY, USA, 2007. ACM.

[18] R. Geambasu, J. Falkner, P. Gardner, T. Kohno, A. Krishnamurthy, and H. M. Levy. Experiences building security applications on DHTs. technical report. UW-CSE-09-09-01.

[19] R. Geambasu, T. Kohno, A. Levy, and H. M. Levy. Vanish: Increasing data privacy with self-destructing data. In *Proc. of the 18th USENIX Security Symposium*, 2009.

[20] R. Geambasu, A. Levy, P. Gardner, T. Kohno, A. Krishnamurthy, and H. M. Levy. Vanish website. http://vanish.cs.washington.edu/.

[21] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report TR2003-1890, Cornell University Computing and Information Science, February 2003.

[22] G. Itkis and L. Reyzin. Forward-secure signatures with optimal signing and verifying. In *CRYPTO*, pages 332–354, 2001.

[23] A. Juels and J. G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, 1999.

[24] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *International Workshop on Peer-to-Peer Systems, (IPTPS)*, 2004.

[25] J. Markoff. New technology to make digital data self-destruct. *The New York Times*. July 20, 2009.

[26] U. M. Maurer. Conditionally-perfect secrecy and a provably-secure randomized cipher. *J. Cryptology*, 5(1):53–66, 1992.

[27] P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems, (IPTPS)*, 2002.

[28] G. Memon, R. Rejaie, Y. Guo, and D. Stutzbach. Large-scale monitoring of DHT traffic. In *IPTPS*, 2009.

[29] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, January 2005.

[30] S. J. Murdoch. Hot or not: Revealing hidden services by their clock skew. In *ACM Conference on Computer and Communications Security*, pages 27–36, 2006.

[31] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *IEEE Symposium on Security and Privacy*, pages 183–195, 2005.

[32] S. J. Murdoch and P. Zielinski. Sampled traffic analysis by Internet-exchange-level adversaries. In *Privacy Enhancing Technologies*, pages 167–183, 2007.

[33] L. Øverlier and P. F. Syverson. Improving efficiency and simplicity of Tor circuit establishment and hidden services. In *Privacy Enhancing Technologies*, pages 134–152, 2007.

[34] M. D. Raimondo, R. Gennaro, and H. Krawczyk. Secure off-the-record messaging. In *WPES*, pages 81–89, 2005.

[35] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *SIGCOMM*. ACM, August 2005.

[36] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[37] M. Steiner, W. Effelsberg, T. En-Najjary, and E. Biersack. Load reduction in the KAD peer-to-peer system. In *DBISP2P*, 2007.

[38] M. Steiner, T. En-Najjary, and E. W. Biersack. A global view of Kad. In *IMC '07: Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 117–122, New York, NY, USA, 2007. ACM.

[39] D. Stutzbach and R. Rejaie. Capturing accurate snapshots of the Gnutella network. In *Global Internet Symposium*, pages 127–132, 2005.

[40] D. Stutzbach and R. Rejaie. Improving lookup performance over a widely-deployed DHT. In *INFOCOM*, 2006.

[41] D. Stutzbach and R. Rejaie. Understanding churn in peer-to-peer networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM.