Hoot

Dustin Bachrach Christopher Nunu
Department of Computer Science
Rice University
Houston, Texas
{ahdustin,canunu}@gmail.com

Dan Wallach
Department of Computer Science
Rice University
Houston, Texas
dwallach@rice.edu

Matthew Wright
Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, Texas
mwright@cse.uta.edu

ABSTRACT

FiXme: *TODO: This is an example of the hlfixme command. When we get down to the last 2-3 days, we'll set the fixme status from "draft" to "final," which will prevent compilation to pdf without fixing or removing all the fixmes.*

1. INTRODUCTION

- Problem: Twitter like semantics w/ encrypted messages - Follow a Hash tag - Take hash tag and create something with crypto strength - Something derived from tags you can search on - But also deliberate collisions (cover traffic)

- Like to have thing that feels like twitter but anonymity properties:

- Twitter/Facebook relevant in Tunisia, (Social media playing big role in revolution across many countries. govt deliberately shut down)

- While we cannot keep them from filtering out service altogether, want to have private communication in plain sight (not steganographic)

- Strong crypto usable by people whispering to each other in streets

- Only trusted channel is not electronic (spoken word), to exchange key.

- Mention how one of our goals is to work within the 140 characters in that we want to fit our protocol as small as possible with as little encryption overhead.

- complimentary to Tor, solve problems Tor+Twitter does not

- What we are doing – Define a protocol for users to communicate over an insecure public network like twitter with message confidentiality and subscriber anonymity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

- Vocabulary

2. THREAT MODEL

Our goal is to allow a user to send a secure message to a private group of individuals allowing only the group members to read the plain-text message. We wish to guarantee as much privacy as possible via an open public timeline like that of Twitter. We now discuss the threat model involved in evaluating the design decisions for the security protocol.

There are two entities which can attack this system. One is the service provider for the communication like Twitter. The other is an active third party observer who is either trying to gather information about what is being said or to whom it is being said.

Imagine an evil Twitter that can maliciously tamper with any of the tweets posted. Since Twitter has full control of the service, they act as an empowered man-in-the-middle between a secure message sender and the recipient group members. Our goals are to prevent Twitter or a third party from reading, creating, or altering a secure message, or discovering who the group recipients are.

Clearly, we cannot directly protect the identity of the secure message sender. A message must be posted to Twitter from some account. By definition, Twitter must know who that user is. We do not consider this limitation a security flaw, since it is fundamental to the problem description. However, if sender anonymity is required, tools like Tor could provide the needed indirection.

Twitter can simply refuse to post a tweet, which is a simple Denial of Service attack. Like any platform, protecting against DOS is nearly impossible. A slight modification to this, is the DDOS, which can be performed by other malicious Twitter users. A malicious user can deliberate tweet hundreds of thousands of messages with the public identifier of a group, which could possibly overload a user's search for the identifier with noise. Later we will describe how we prevent against this action and in fact utilize it to provide added security.

An attack that can be administered by Twitter or a third party is a replay attack. Since Twitter is a public forum, anyone can see the encrypted texts posted, and nothing prevents someone from simply copying such a message and posting it again. We want to protect against this threat.

A malicious Twitter also can take a valid Hoot and change the

associated author or time that it displays with it. Our system does not directly prevent against these attacks, but they can be addressed by including the time stamp and author name with in the plain text of the message.

Finally, we want to prevent Twitter or any third party from gathering trending data on the encrypted posts. Even if they cannot read the messages, they can observe that someone is posting to the same group of recipients following certain patterns (such as daily at X time, or after major events), this repetition could compromise the group's identity by essentially creating a profile. This type of information is less of a concern to users of Twitter that are simply chatting, but one can imagine that this information can end up being used to identify spies. For example, by observing a pattern between company secrets being leaked to some competitor and a particular employee's secret tweets to a group just hours prior to each incident, the leaker could be identified. This attack can be prevented by using anonymizing services like Tor coupled with a collision technique we will describe later to create "cover traffic."

- Punting on plain tag distribution - Does not protect against double agents. (known weakness) - Key should be sharable by gossip but gossip is also weakness. - Crypto strength to be just enough that its easier to attack via gossip than via crypto

-omnipotent eavesdropper

- assume "whisper channel"

- Why Johnny Can't Encrypt - Paper

3. BACKGROUND

1) History of government censorship, man in the middle - Tunisia code injection - Chinese firewall - Crypto keys for important services to iranian source (Komodo) - Person providing netwrok (even over ssl) might be evil

2) Tor - Trying to work against government censorship

- Group Crypto Keys

3.1 Definitions

Privacy for groups and individuals in those groups has been investigated in a variety of contexts. In this section, we describe how the aims of Hoot relate to those of previously studied systems.

Anonymity.

Systems for online anonymity, like Tor [**?**] and Mixminion [**?**], aim to protect users from being linked with their traffic.

In this context, we can consider a variety of privacy attributes, including:

- *sender anonymity*: The sender of a message is not identifiable from among a set of possible senders.
- *recipient anonymity*: The recipient of a message is not identifiable from among a set of possible recipient.
- *unlinkability*: Any of the items of interest (senders, recipients, or messages) cannot be linked with other items of interest.
- *pseudonymity*: The use of pseudonyms as identifiers, either for a sender or a recipient.

Pfitzmann and Hansen have compiled a rich discussion of the meaning of these and other terms [**?**]. Unlinkability is a rather broad term. Two useful terms that can be derived from it are *recipient unlinkability* — the recipient cannot be linked with the sender(s) and messages — and *relationship anonymity* — the sender and recipient cannot be linked with each other.

Hoot does not seek to provide sender anonymity: the adversary can observe the fact that a given sender sent a particular message. Against weaker adversaries who cannot eavesdrop on the sender, Hoot can provide pseudonymity. More importantly, Hoot does aim to provide recipient

It does

Anonymity: - sender anonymity - Unlinkability - Receiver anonymity - subscriber anonymity - Relationship anonymity - Group anonymity

Social network privacy - Community privacy - Adversarial community discovery

Data and database privacy

Anti-censorship mechanisms: - Eternity, Freehaven - Freenet, Gnunet - Publius, Tangler, Dagster - Perng - Serjantov

Metrics - k-anonymity - l-diversity - differential privacy - plausible deniability -

Techniques: - Cover traffic - Link cover - Path cover - RBC - PIR/OT

4. DESIGN

The Hoot protocol consists of a message header and a message body. The header contains the identifier, Encryption Key, MAC Key, and integrity digest. Messages are indirect in that they do not reference recipients. Anyone who knows the shared secret must be able to read the message, but should also be able to find the message easily. Rather than attempting to treat every message posted to Twitter as a potential group message, and thus decrypting the entire Twitter stream, the protocol places an identifier into every Hoot so a fellow group member can simply search for the identifier to see all potential messages. With a constant group identifier, readers can also publicly follow that identifier like any other Hash Tag on Twitter.

To create an identifier, we must find a short set of bits that are derivable from the shared secret, but give an attacker little information about the shared secret itself. Hash functions provide a great way to get a set of bits from a shared secret without divulging much information about the original shared secret. In our protocol, the shared secret is called a *Plain Tag*, which is comparable to a Twitter Hash Tag. The result of hashing the Plain Tag with a given hash function, H, is referred to as the *Long Tag*. Thus,

$$LongTag \leftarrow H\,(PlainTag)$$

The protocol could simply use the Long Tag as an identifier, but this choice leads to several problems. First, to uphold our design goal of being concise and to fit within Twitter's 140 character limit, it is infeasible to use the full output of a hash function. Secondly, strong hash functions do not produce many collisions. If a group is communicating in secret, not only do we want to protect the content of the communication, but we would like to conceal subscription to a particular group. For example, a rebellion group wishes to communicate over Twitter using Hoots, but it can be dangerous for a supporter of the rebellion to listen and subscribe to the identifier of the group. If, however, the identifier of the group can collide with the identifier of a popular Internet topic, like Charlie Sheen, group followers can shadow their rebellious activities with other innocent topics.

To generate a collision, we need to shorten the long tag, generating a *Short Tag*, which will induce more collisions. The shorter the Short Tag, the higher the collision rate will be and the less sure an observer can be of what topic a Hoot reader is listening to. With this greater anonymity comes more computational work, however. Since more group messages will now be belong to the same identifier, a follower must decrypt more messages to find relevant sig-

nal. Depending on the required degree of subscriber anonymity, more collisions might be worth the computational overhead. Also, even given a constant Short Tag length, a group can choose a tag that will collide with extremely popular tags to generate even more noise. Given a Long tag of byte-length *N*, we produce a Short Tag by taking the first *K* bytes of it:

$$ShortTag \leftarrow LongTag_{[0:K]} : K < N$$

The header also contains an set of encrypted session keys, *(Encryption Key, MAC Key)*. To combat replay attacks, the protocol uses a random session key to encrypt the plain text. This session key, called the Encryption Key, is then included in the set of keys in the header. The protocol ensures integrity the standard way by including an HMAC of the cipher text (http://www.daemonology.net/blog/2009-06-24-encrypt-then-mac.html). The key to the HMAC function is randomly generated and is included in these set of keys:

$$EncryptionKey \leftarrow randomBytes\,()$$
$$MacKey \leftarrow randomBytes\,()$$

To encrypt this set of keys, we exploit the entropy of the Long Tag. Since the identifier for a message is only the first *K* bytes in the Long Tag, we can use the latter set of bytes elsewhere. By encrypting the keys with part of the Long Tag, a reader can easily know the key to decrypt with since it is simply the hash of the Plain Tag, and we also do not require knowledge of two different secrets. Thus, given an encryption function and corresponding key, E_{key}, we have the encrypted keys as:

$$SessionKeys \leftarrow E_{LongTag_{[J:N]}}\,(EncryptionKey, MacKey) : K < J \leq N$$

The body of the Hoot is the cypher text, *C*, of a message, *M*, encrypted with the Encryption Key:

$$C \leftarrow E_{EncryptionKey}\,(M)$$

Both encryption processes do not need an Initialization Vector to prevent replay attacks. The encryption operation to generate the *SessionKeys* uses a constant key, but the content which it encrypts is two random stream of bytes. Even with the constant key, it will encrypt to a different cipher-text everytime. The encryption operation to generate C uses a random session key, *EncryptionKey*, which will also produce different cipher-text for the same message. This shows why we do not need an IV and that our protocol is safe from replay attacks. A receiver can keep track of all the session keys it has previously seen, so if it encounters a new message with keys it has already seen and the same exact message, it can dismiss the message as a replay.

The header also contains the integrity digest, *D*, of the cypher text, using the MAC Key in the header. By including an integrity digest, the protocol allows for quick verification that a message is for a specific group without having to decrypt the message. It also verifies that the Hoot has not been tampered with.

$$D \leftarrow HMAC_{MacKey}\,(C)$$

Therefore, a complete Hoot appears as:

$$Hoot \leftarrow ShortTag, SessionKeys, D, C$$

5. IMPLEMENTATION

In this section we describe an implementation of the Hoot protocol we create for our prototype.

5.1 Generating a Hoot

The Hoot protocol can be implemented in a variety of ways using different encryption and hashing algorithms. We created a Python prototype that fully implements the protocol.

A Long Tag is generated by running a SHA256 hash over the Plain Tag. The first 128 bits of the Long Tag are dedicated to identification. We then take the first m bytes of the Long Tag to get a Short Tag, where m is the desired Short Tag length.

The Encryption Key is 128 random bits, and the MAC Key is 160 random bits. These random numbers are concatenated together and then encrypted using AES with the last 128 bits of the Long Tag as the key.

The plain text is encrypted using AES with the Encryption Key as the key. An integrity check is created by performing an HMAC-SHA1 of the cipher text and the MAC Key as the key.

With all components created, a Hoot can be generated by printing out a # symbol, the short tag, a space, the encrypted keys, the HMAC digest, and the cipher text.

5.2 Message Length

Given our goal to conform to Twitter, the final output of a message must be under the 140 character limit. Another goal is to be concise and use as little overhead as possible for the entire encryption process. Twitter has a very broad definition of a character. It is not simply ASCII characters, but UTF-8 as well. Based on the encoding, we can squeeze the encryption into even fewer characters. However, this manipulation has issues since many clients do not conform to UTF-8, and there will be little gain if the plain text itself is written in UTF-8. Our prototype can generate Base64 encoded messages or Unicode messages.

Base64 encoding is simple, universal, yet yields longer message sizes. Given a short tag of length m and a plain text message of length n, a Base64 encoded message has

$$m + 20 \times \mathit{floor}\left(\frac{n}{16}\right) + 118$$

characters. Assuming a Short Tag of 2 characters, to fit a Base64 encoded message into 140 characters, the plain text message can be at most 31 characters.

A unicode message on the other hand can be much shorter. Given m and n, a Unicode encoded message has

$$m + 12 \times \mathit{floor}\left(\frac{n}{16}\right) + 64$$

characters. Assuming again a Short Tag of length 2, a Unicode encoded message can fit within 140 characters when the plain text is at most 111 characters. By using Unicode, the Hoot protocol only uses 29 characters of overhead and can handle reasonable length messages. Obviously, if longer messages are needed, multiple Hoots can be posted, but by having such a low overhead, we are getting 79% utility out of the 140 characters.

6. EXPERIMENTS

We performed the following experiments using our prototype to validate the protocol.

6.1 Cover Traffic

The power log scale shows us that there is a large spectrum of subscriber anonymity that can be taken advantage of.

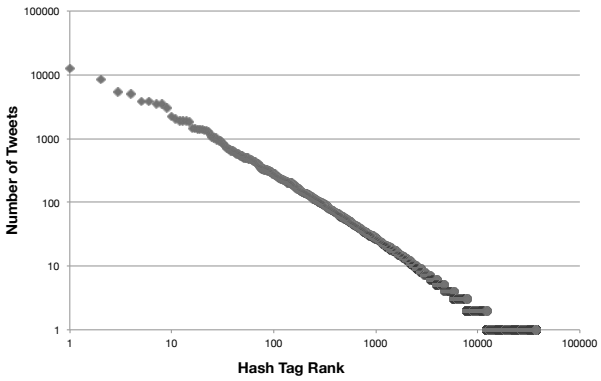

Figure 1: 2009 Twitter Hash Tag distribution on a log-log scale.

Suffix Length (characters)

Short Tag Length (characters)

	3	4	5
2	0.024336200%	0.025141500%	0.024525600%
4	0%	0.000013535200%	0.0000068767300%
8	0%	0%	0%

Figure 2: Percent of search space that returned hits for an entered Short Tag, with a Prefix of 'rice', and given Suffix Length.

–How long a short tag should be? Should we break down log-log scale further, or simply examine average tag length?

size 8 seems to eliminate most collisions, allows for exact look up should you choose it.

6.2 Collider

In order to explore the feasibility of steering a particular Plain Tag to collide with an existing Short Tag, we built a collider tool. The tool takes in a Short Tag substring, *T*, a prefix string, *S*, a suffix length, *L*, an alphabet *A*. The collider finds all concatenations of *S* and strings of length *L* from *A** such that the truncated hash of the resultant string matches *T*.

The search space, then, is of size $|A|^L$. If we wanted to match byte strings, $|A| = 256$, however, we decided to restrict the alphabet to alphanumeric characters, yielding $|A| = 62$. Additionally, the search space can be explored in parallel, giving the Collider executing on a system with *P* processing units, a runtime of $O(\frac{62^L}{P})$. (this is incredibly parallel so easier to bruteforce)

As expected, the runtime in Figure 3 scales exponentially with *L*.

- what that tells us about design parameters of solution
- Then we can conclude that we need this number of bits, etc
- Wrap up all experiments into What can our system do
- Easy to collide or not to collide (Tune cover traffic)

6.3 Performance

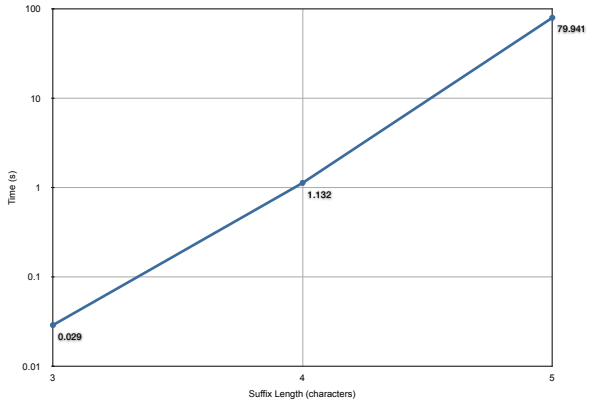

Figure 3: Runtime for the Collider to search the exploration spaces of sizes 62^3, 62^4, 62^5 on dual quad-core Intel i5 processors.

Table 1: Average Hoots Per Second for Encryption and Decryption

Action	Average Hoots per second
Encryption	3614.531
Decryption	15587.328

In this experiment we wanted to see how well the encryption engine performed. In March 2011, Twitter stated that the site receives 140 million tweets per day or 1620 tweets per second on average (http://blog.twitter.com/2011/03/numbers.html). They also said that the maximum tweets per second ever was 6939. These numbers act as rough upper bounds to the number of Hoots per second the system would need to keep up with. In all likelihood, the number of encrypted messages posted would be drastically smaller than regular messages.

The benchmarks in Table 1 were run on a 1.86GHz Intel Core 2 Duo, 4GB Ram, Macbook Air using Base64 encoding.

Given these numbers, a Twitter server could easily encrypt the entire Twitter feed as the messages were posted. To handle peak usage like the 6939 tweets per second Twitter observed, many optimizations could be made, the simplest being to add a couple computers to help with the load. Our experiment shows that the Hoot protocol does not have significant overhead during encryption or decryption, so it can be adopted with little engineering effort.

It is interesting to note that the decryption rate is important to a client, since a client will be searching tweets trying to identify and decrypt potential Hoots. A client may not trust Twitter to do the decryption since that involves sharing the Plain Tag with Twitter, so a client would decrypt the message on their machine. Almost every client will not have a datacenter of computers for decryption, but our process can decrypt Hoots almost five times faster than it encrypts them. A client, even with limited computing power, can easily keep up with the Twitter feed.

7. DISCUSSION

In this section, we discuss a variety of issues and future extensions of the Hoot design.

7.1 Incremental Rollout

1. Can you incrementally roll it out - Service provided by twitter or yourself 2. 140 character limit - Twitter doesn't have to go too far to have all the metadata for encryption

– How hard is it for twitter to do this. - Reference message fitting into 140 char using unicode - Reference peak hps, and how our system holds up

7.2 Adoption

The next question to ask after we have shown that rolling out the Hoot service is feasible would be whether Twitter would actually implement such a feature. Based on the nature of Twitter, we believe that Twitter would not add a secure messaging framework like Hoot. Twitter as a company needs to know what people are talking about, so it can provide relevant advertisement. Adding the Hoot infrastructure to Twitter would prevent Twitter from knowing the content of the messages, and so we believe the Hoot service will never be adopted. However, this need not prevent individuals from using the Hoot protocol over Twitter. As long as Twitter faithfully delivers tweets, users are free to run the Hoot encryption on their own machines over their messages and then post the output to Twitter. In fact, this method is more secure in that a user only has to trust their machine. If Twitter is responsible for encrypting a message, nothing is stopping them from keeping a copy of the plain text message. Paranoid users will always want to encrypt messages themselves, so Twitter adopting the Hoot protocol is unnecessary.

7.3 Usability

As described in the Cover Traffic section, a group can deliberately collide with a popular tag by concatenating an easily rememberable string of text with random letters or numbers. As Miller[**?**] noted, people can remember 7 +/- 2 unique packets of data, so as long as these collisions can be generated with a suffix in that range a user is likely to be able to remember it. We could further improve rememberability by restricting suffix values to only digits, and then generate a suffix of 7 or 10 digits, emulating a phone number. The main requirement is that a group should be able to have a unique shared secret that can deliberately collide with other tags while still being easy to remember and more importantly easy to transfer. Since our proposal does not deal with key transfer, we assume our key communication is done through whisper channels and thus must be short and memorizable for simple transfer. We have found that a collision can be found with only appending 3-6 characters to a prefix, so deliberate collisions can both be found and transferred without much effort.

– Entropy 7+-2 – Deliberate posting crap (to hide your stuff). – Increases work factor (signal to noise) – Fairly strong attacker, but user who don't have ability for a lot of entropy – 70 bits is enough compared to the weakness of everything else – Discuss how we can make it harder for an attacker. Maybe hashing 1000 times...

7.4 Adaptability

Since Hoot benefits from and encourages collision, there will come a point when there is too much collision, i.e. there will be too much noise to signal for a specific group when following an identifier that collides with many groups. At some point, the entire Hoot universe will want to decrease the amount of collisions. The simple solution is to increase the Short Tag length. Whenever there is too many collisions, the Short Tag length will be increased by one, which will result in far less collisions until the communication increases and eventually the cycle continues.

The nature of Hoot makes it convenient for this Short Tag length tuning to be done at the system level or at the group level. For instance, a group that is overrun with collisions with popular group could simply increase their Short Tag length internally. All Hoots would include the extra character in the Short Tag, and all subscribes would search for the longer Short Tag. It is also as simple for the entire system to switch to a longer identifier by having an "oracle" broadcast to everyone that the system Short Tag length has increased.

Even in a distributed system where some users may not receive the oracle's broadcast, the system still functions. For example, assume user A is following a group who's Long Tag begins "A5trxq...", and in A's view, Short Tags should be 2 characters. For user B, who posts messages to the group, the Short Tag length should be 3. B's Hoot would then look like "#A5t", but A would be searching for "#A5". Twitter's search functionality will still return B's messages for A's search. Although A will be getting much more noise than signal when they search at the shorter length, they will not miss any Hoots.

Previously, we stated that a Short Tag is the first K bytes of the Long Tag, so we explicitly put a limit onto how long a Short Tag can be. Does this prevent us from adapting to an ever growing Hoot ecosystem? In our implementation, we use the first 16 bytes of the Long Tag for identification. Therefore using Base64 encoding, we get 62^{16} or 4×10^{28} unique identifiers. In a system with 16 byte Short Tags, there would rarely if ever be collisions, much less too many collisions to increase the length.

Finally, as a result of the encoding systems, the size of the Short Tag space increases by a large factor for each character added. This takes a space that is highly populated to a space that is very sparse, so it is very possible that by adding one character to tag identifiers that groups would be uniquely identified. Groups that also deliberately collided with another group would often not collide at a longer Short Tag length. It would be convenient to instead grow the space by single bits at a time, so as to double the space, a much smaller factor. Unfortunately, Twitter is a character driven environment rather than bit driven, so this is difficult to address. It also does not deal with deliberate collisions. All deliberate collisions will need to be regenerated after a Short Tag length increase.

chris - need a citation for core growth so we can make argument about whether searching will keep being feasible - twitter vs moore - need a citation, or argument, for how fast we expect twitter to grow in the future, can we turn the nob slow enough to keep in pace with processing power, and still be usable?

7.5 Alternate Backends

Even though our protocol was designed with Twitter in mind, it is extensible to other systems and platforms. The Hoot protocol describes a secure way to transfer short messages (with short encryption overhead) across a publicly available network completely openly. Twitter is a great example of this environment but not the only one. Hoot could interact, for example, with a Distributed Hash Table system (DHT) like CHORD or Pastry. Importantly, Hoot can interface with both centralized systems like Twitter or Buzz or decentralized systems. As long as the platform's content is publicly searchable, the Hoot protocol allows secure transmission to anonymous group followers.

8. CONCLUSIONS

FiXme: *TODO:*

9. REFERENCES