Fighting Fake News in Encrypted Messaging with the Fuzzy Anonymous Complaint Tally System (FACTS)

Linsheng Liu*, Daniel S. Roche†, Austin Theriault*, Arkady Yerukhimovich*
*George Washington University *lls,atheriault,arkady@gwu.edu*
†United States Naval Academy *roche@usna.edu*

***Abstract*—Recent years have seen a strong uptick in both the prevalence and real-world consequences of false information spread through online platforms. At the same time, encrypted messaging systems such as WhatsApp, Signal, and Telegram, are rapidly gaining popularity as users seek increased privacy in their digital lives. The challenge we address is how to combat the viral spread of misinformation without compromising privacy. Our FACTS system tracks user complaints on messages obliviously, only revealing the message's contents and originator once sufficiently many complaints have been lodged. Our system is *private*, meaning it does not reveal anything about the senders or contents of messages which have received few or no complaints; *secure*, meaning there is no way for a malicious user to evade the system or gain an outsized impact over the complaint system; and *scalable*, as we demonstrate excellent practical efficiency for up to millions of complaints per day. Our main technical contribution is a new *collaborative counting Bloom filter*, a simple construction with difficult probabilistic analysis, which may have independent interest as a privacy-preserving randomized count sketch data structure. Compared to prior work on message flagging and tracing in end-to-end encrypted messaging, our novel contribution is the addition of a high *threshold* of multiple complaints that are needed before a message is audited or flagged. We present and carefully analyze the probabilistic performance of our data structure, provide a precise security definition and proof, and then measure the accuracy and scalability of our scheme via experimentation.**

I. Introduction

The proliferation of fake and misleading information online has had significant impact on political discourse [23] and has resulted in violence [37]. Large services like Facebook and YouTube have begun to remove or label content that they know to be fraudulent or misleading [1], [2], through a combination of a manual process of reviewing posts/videos and automated machine learning techniques.

However, on end-to-end encrypted messaging services (EEMS), like Signal, WhatsApp, Telegram, etc., where so-called "fake news" is also shared, such review is impossible. At no point do the providers see the plain-text, unencrypted contents of messages transmitted through their systems and thus cannot identify and remove offending material. Such platforms must instead rely on their users to identify and report malicious content. Even then, identifying and removing *users* who repeatedly post misleading and dangerous content may still be difficult because some platforms, like Signal, also hide the path the message took, so identifying and addressing the original source of the misinformation may not be possible.

Tyagi et al. [43] introduced a first approach for overcoming this challenge and allow EEMS to effectively *traceback* an offending message to find the originator based on a user complaint. The traceback procedure also assures that all other messages remain private and that innocent parties cannot be blamed for originating the offending messages.

While innovative, there are two notable shortcomings of Tyagi et al.'s traceback scheme. First, it requires extensive "housekeeping" on the part of the platform that scales as the number of messages in the system. Second, a single, possibly malicious, complaint can trigger a traceback and thus reveal the message contents as well as the history of prior recipients, which is counter to the goals of EEMS to maintain the privacy of users communicating through this system. One malicious user (e.g., a government agent) can reveal the source of a piece of information (e.g., a leak) that they have received, violating the privacy of the sender (e.g., the leaker) by issuing a single complaint to the EEMS. While it may be possible to apply manual review to these complaints, the scale of possible complaints could make this impractical. Several follow-on papers [24], [36] show how to achieve *source-tracking* for EEMS to identify the source of a message without relying on traceback of all intermediate recipients. However, these systems still allow a single complainer to trigger the source-tracking.

In this paper, we aim to resolve this conflict between privacy and ability to identify misinformation in EEMSs by first observing that "fake news" messages are, by definition, viral and are thus received, and likely complained about, by a large number of users. Private messages, such as leaks, on the other hand, are likely to be targeted and are thus only received by a small number of users; indeed, any message received by only a few users is inherently less impactful overall and more likely deserving of privacy protections. This leads to a more nuanced approach for identifying fake news: apply a threshold approach to complaint management, whereby only viral fake

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23109
www.ndss-symposium.org

news would overcome the threshold and trigger an audit.

Counting the number of complaints in a private manner is a non-trivial problem if the privacy of the EEMS' clients is to be maintained prior to the threshold being reached, even given available cryptographic solutions. For example, a homomorphic encryption solution (e.g., [26]) would enable the checking and updating of counts for each message, but the access patterns of clients checking and updating counters could reveal how many complaints a message receives even if the threshold is not reached. Oblivious RAM (ORAM) (e.g. [20], [40]) could be used to protect the access patterns, but have high computational overheads and usually assume clients may share secrets and are not malicious. Pricate Information Retrieval (PIR) does not assume clients are trusted, but has different scalability challenges and does not address the problem of obliviously updating without revealing which message is being complained about.

We propose a different approach we call a Fuzzy Anonymous Complaint Tally System (FACTS). FACTS maintains an (approximate) counter of complaints for each message, while also ensuring that, until a threshold is exceeded, the status of these counters is kept private from the server and all users who have not received the message. FACTS builds on top of any end-to-end encrypted messaging platform, incurring only small overhead for message origination and forwarding. In particular, FACTS maintains the communication pattern of the underlying messaging system, requiring no new communication or secrets between users even for issuing complaints.

To avoid the high overheads of existing solutions, FACTS uses a novel oblivious data structure we call a *collaborative counting Bloom filter* (CCBF). This data structure allows us to obliviously increment and query approximate counters on millions of messages while only requiring 12MB of storage. Moreover, incrementing a counter only requires flipping *one bit* on the server and only uses the minimal communication of $\log |T|$ bits to address a single bit in the server-stored bit vector T. While the resulting counters are only approximate, we show experimentally and analytically that we are able to enforce the threshold on complaints with good accuracy, namely, below 10% error in theory, and below 3% in most realistic deployment scenarios.

The contributions of this paper are as follows:

- We develop a collaborative counting Bloom filter, a new oblivious data structure for counting occurrences of a large number of distinct items.
- We use this data structure to instantiate a provably-secure system, FACTS, for privacy-preserving source identification of fake news in EEMSs.
- Finally, we perform experiments to show the accuracy and overhead of FACTS in realistic deployment scenarios.

A. Setting and Goals

FACTS is built on top of an end-to-end encrypted messaging system (EEMS). For this work, we focus on the setting of server-based EEMSs with a server S that enables (authenticated) encrypted communication between the system users. Examples of such EEMSs include Signal and WhatsApp, among many others.

To make sure that FACTS is compatible with existing encrypted messaging systems, we make the following performance requirements:

1) **Messaging costs:** Originating and forwarding messages should incur little computational overhead for both users and the server over the standard procedure in the encrypted messaging system,
2) **Server storage:** The storage overhead of the server should be small (i.e., a single table not exceeding a few MBs),
3) **User costs and requirements:** Issuing complaints requires a small amount of communication and computation from the complaining user, and no cost to other users. Moreover, complaints can not require direct communication between users or require the users to have any apriori shared secrets that are not known to the server.
4) **Complaint throughput:** Issuing complaints may be slower than standard forwarding of messages, but the system must be able to handle millions of complaints per day.

To ensure privacy of messages and complaints, FACTS requires that complaints remain hidden from the server (and colluding clients) until a threshold of complaints is reached. Additionally, FACTS ensures integrity of the complaint process ensuring correctness of complaint counts and the identity of the revealed originator once the threshold is reached. Specifically, FACTS satisfies the following security guarantees:

1) **Message privacy:** All messages remain end-to-end encrypted and private from the server and non-receiving clients until a threshold of complaints is reached and an audit is issued. Moreover, even after the audit, only information about the audited message is revealed.
2) **Originator integrity:** Once a threshold of complaints is reached on a message, FACTS will only identify information about the true originator of the message. In particular, no innocent party can be framed as the originator.
3) **Complaint privacy:** The server and any colluding clients who have not received a message x should have no information about the number of complaints on x. In particular, the server should not be able to tell what message is being complained about.
4) **Complaint integrity:** A set of malicious clients should not be able to alter the number of complaints on any message x. Specifically, they cannot block or delay complaints, and cannot (significantly) increase the number of complaints on a message x except through the legitimate complaint process.

B. Building FACTS

Recall that our goal is to enable privacy-preserving counters to tally complaints on each message m. This suggests an

immediate solution where the server stores an encrypted counter for each message, and clients interact with the server to increment the counter and check the threshold. While implementing such counters is certainly possible using homomorphic encryption [16] or standard secure computation techniques [4], [19], [46] , the problem is that the access pattern of clients' updates to counters leaks information to the server by revealing the complaint histogram. This suggests a further modification to store the counters inside an oblivious RAM (ORAM) [20] to hide such access patterns from the client. However, in our setting this would require a multi-client ORAM [7], [22], [30] which incurs significant performance penalties including at least $O(\log n)$ communication overhead when there are n distinct messages. Moreover, this would require direct communication between clients to maintain their ORAM state, and additionally, no security against malicious clients.

In FACTS, we take a different approach. Instead of relying on encryption to hide the counters from the server, we hide the counters in plain sight by mixing together the counters for all the messages in a way oblivious to the server. To make this possible, we relax the functionality of FACTS to only enforce approximate, rather than exact, thresholds. That is, the threshold will be triggered on a message x after $(1 \pm \epsilon)t$ complaints for a small error ϵ. Making this relaxation allows us to use a sketch-based approach for counting the complaints.

To achieve this functionality obliviously, we develop a collaborative counting Bloom filter (CCBD). This data structure consists (roughly) of a collection of Bloom filters, one for each message, where the Bloom filters corresponding to different messages are mixed together to hide them from the server. Specifically, the server stores a table of s bits. A random subset of v bits (V_x) is assigned to each message x at origination; these bits will be used for tracking complaints about this message (for intuition, one can think of these bits as forming a Bloom filter for storing the set of complaints about the message). We stress that the server has no information about which bits correspond to which messages.

To complain about a message x, a user who has received x can find the corresponding bit locations, and will (attempt to) flip one of the bits from 0 to 1. However, allowing users to flip any bit they choose, would allow malicious users to significantly accelerate complaints for a message they wish to disclose. To prevent this behavior, we restrict each client to only be able to flip (i.e., complain on) a small (of size u) set of locations U_C. Thus, to complain about a message m, a client first identifies the set V_x of bits corresponding to x. Then, she checks how many of these bits have already been set to 1, and if this exceeds a specified threshold, notifies the server to trigger an audit. If the threshold for x is not yet exceeded, the client sees whether any of the 0 bits in V_x are in her set U_C, and if there are any such bits, she flips one of them (chosen at random) from 0 to 1. Otherwise, the user still flips a random bit in their set U_x, so the server cannot discern anything about the message being complained on. We prove in Section IV that the actual number of complaints necessary to trigger an audit can be calculated with high precision allowing us to (approximately) enforce the desired threshold.

C. Limitations of FACTS

In order to present FACTS, it is also important to recognize what our system does *not* do.

First, unlike some prior work, e.g. [15], [28], FACTS does not attempt to automatically detect misinformation. Instead, it relies on users reporting it when they see it. This reliance on users has inherent benefits and limitations. While our system is not subject to the kinds of machine-generated false positives that can arise from, e.g., hash collisions [5], our model is inherently vulnerable to any sufficiently large group of dishonest users, who could trigger an audit on a benign message. This is why we suggest the possibility of a manual human review process on message contents before the service provider would take any action on an audited message; see Section VIII.

Second, due to the approximate nature of FACTS, it works most effectively for relatively large thresholds, say in the hundreds and above. For our application to fake news detection, this is reasonable as such messages are likely to garner a large number of complaints, and indeed this was our main motivation for this paper. We leave as interesting possible future work to implement a system supporting smaller thresholds, even as small as 2, efficiently.

One additional functionality limitation is that, as is true with any application using Bloom filters, the CCBF data structure can fill up once too many complaints have been registered. To deal with this issue it is necessary to periodically reset the counters and refresh the CCBF data structure. We refer to each such refresh period as an *epoch*, and in the remainder of the paper only present algorithms for a single epoch.

Finally, on the security side, an important limitation is that FACTS reveals meta-data on who issues complaints (but not what message they complain on). It is important to consider what is revealed by this meta-data. By observing the timing of messages and complaints, the server can make some inferences about what messages users are sending and complaining about. For example, suppose that the server sees that A sends a message to B, and then B issues a complaint. Then, it may be reasonable for the server to assume that A has sent the message which B complained about, even though this is not directly leaked by our system. Nonetheless, our definition guarantees that the server cannot be certain that this is indeed the case. We note that the messaging meta-data is already a byproduct of the underlying EEMS platform. FACTS only adds complaint meta-data to this leakage; see Section VIII for some further discussion.

D. Paper Layout

The remainder of the paper is organized as follows. In Section II, we introduce some of the notation we use throughout the paper. Then, in Section III we describe the syntax and functionality of FACTS. In Section IV we present and analyze our main building block, the CCBF data structure. Then, in

Section V, we show how to use a CCBF to instantiate FACTS. We demonstrate the accuracy and performance through experimental evaluation in Section VI and then prove the security of FACTS in Section VII. Finally, we describe some variants of FACTS and directions for future work in Section VIII and present related work in Section IX.

II. Preliminaries

We use $[n]$ to denote the set $1, \ldots, n$. We write $x \leftarrow X$ to indicate that the value x is sampled uniformly at random from the set X. We use λ to denote a statistical security parameter and κ to denote a computational security parameter. We also assume the existence of a hash function $H : \{0,1\}^* \to \{0,1\}^*$ which is modeled as a random oracle. We let $\mathsf{poly}(\cdot)$ denote a polynomial function and $\mathsf{negl}(\cdot)$ denote a negligible function.

III. Fuzzy Anonymous Complaint Tally System (FACTS)

In this section, we present the syntax for FACTS and describe how FACTS is used. We show how to instantiate FACTS in Section V.

Assumptions: We assume that each user A has a unique identifier ID_A, and that the server can authenticate these IDs. (We will abuse notation to use A to represent the user and also the id ID_A). We also assume that the server has an identifier ID_S (we will denote this by S) that can be authenticated by all users.

Additionally, we assume that the underlying end-to-end encrypted messaging system (EEMS) offers methods $\mathsf{send}(A, B, x)$ and $\mathsf{receive}(A, B, x)$ for sending and verifying a message x sent from user A to user B. Moreover, we assume that this communication is encrypted and authenticated. In particular, receive verifies that the received message was sent by A and was not modified in transit. Importantly, we do not assume that this platform is anonymous, instead assuming that the full messaging history i.e., who sent a message to whom and the size of that message, is available to the server.

Syntax: FACTS is a tuple of protocols $\mathsf{FACTS} = (\mathsf{Setup}, \mathsf{SendMsg}, \mathsf{RcvMsg}, \mathsf{Complain}, \mathsf{Audit})$. The first is used to set up FACTS, the next two are used to send and verify messages, while the last two methods are used to issue complaints and audit received messages.

- $\mathsf{Setup}(c)$: This takes as input an upper bound on the total number of users and initiates the FACTS scheme for c users.
- $\mathsf{SendMsg}(A, B, \mathsf{tag}_x, x)$: This method is used by a user A to send a message x to another user B. This may be a new message *originated* by A (indicated by $\mathsf{tag}_x = \perp$) or a forward of a previously received message.
- $\mathsf{RcvMsg}(A, B, \mathsf{tag}_x, x)$: This algorithm is run by B upon receiving a message (tag_x, x) from A.
  This algorithm checks whether tag_x is indeed a valid tag generated by A on message x. If this is the case, then B accepts the message, otherwise he rejects the message.
- $\mathsf{Complain}(C, \mathsf{tag}_x, x)$: This protocol is run by a user C to complain about a received message (tag_x, x).
- $\mathsf{Audit}(C, \mathsf{tag}_x, x)$: This protocol issues an audit of a message x revealing (tag_x, x) to S. This will be called by C when the number of complaints on m exceeds a pre-defined threshold (with high probability).

Usage: The following workflow demonstrates the standard usage of FACTS. To originate a new message x, a user A runs the $\mathsf{SendMsg}$ protocol with the server S to create metadata tag_x. $\mathsf{SendMsg}$ then sends this metadata and the message (tag_x, x) to the receiving user B using the messaging platforms send method. Upon receiving a message (tag_x, x), B first locally runs $\mathsf{RcvMsg}(A, B, \mathsf{tag}_x, x)$ to verify that the received message and tag are valid, if this fails he ignores the message. To forward a received message (tag_x, x), a user A runs $\mathsf{SendMsg}$ with the server S to produce metadata tag'_x; A then discards this metadata, and the original message (tag_x, x) is sent instead using the messaging platform's send method.[1]

If a user B receives a message (tag_x, x) that it considers "fake", he can use the $\mathsf{Complain}$ protocol to issue a new complaint on this message. After issuing a complaint, B checks whether the threshold of complaints on x has been reached. If so, he calls Audit to trigger an audit on the message (tag_x, x), revealing x and the originator of x to the server S.

We note that users may join and leave during the execution of FACTS as long as the total number of identifiable users does not exceed c.

IV. Collaborative counting Bloom filter

Our system records complaints in a special data structure which we call a *collaborative counting Bloom filter*, or CCBF. This data structure shares some of the same basic functionality as a counting Bloom filter [13], [33] or count-min sketch [9], which is to insert elements and compute the (approximate) frequency of a given element.

Our CCBF differs from a usual count-min sketch in that each update operation is accompanied by a *user id*, and each user can only perform a single update for a given element. This can be thought of as a strict generalization of the normal count-min sketch operations, where the latter may be simulated by our CCBF by choosing a unique user id for each update.

The actual data structure for the CCBF is also far simpler than the 2D array of integers used for a count-min sketch; instead, we store only a single length-s bit vector T. As a result, our CCBF will have the following desirable properties:

- The bit-length of T scales linearly with the total number of insertions.
- Each witness operation (insertion) changes exactly one bit in the underlying bit vector from 0 to 1.
- The CCBF is *item-oblivious*, meaning that after observing an interactive update protocol, the adversary learns which user id made the update, but not which item was updated.

The downside to our CCBF is a far lower accuracy of the count operation in general compared to count-min sketches. However, we will show that, for careful parameter choices,

[1] We note that since the underlying messaging scheme is encrypted, the actual ciphertext sent will not be the same as the ciphertext received.

the count operation is highly accurate within a certain range, which is precisely what is needed for the current application.

A. CCBF Construction

The CCBF consists of a single size-s bit vector T and two operations:

- $\mathsf{Increment}(x, C)$: Increases the count by 1 for item x according to user id C.
- $\mathsf{TestCount}(x, t)$: Returns true if the number of increments performed so far for item x is *probably* greater than or equal to t.

Note that TestCount is probabilistic, in the sense that it may return false when the actual count is greater than t, or true when the actual count is less than t. Our construction guarantees the correctness probability is always at least $\frac{1}{2}$, and our tail bounds below show the correctness probability quickly goes towards 1 when the actual count is much smaller or larger than t.

The performance and accuracy of the CCBF is governed by three integer parameters s, u, and v, with $u, v \leq s$, which must be set at construction time. The first, s, is the fixed size of the table T. Each user i is randomly assigned a static set of exactly u locations in the T; i.e., a uniformly random subset of $\{0, 1, 2, \ldots, s-1\}$, which we call the *user set*. Similarly, each possible item x is assigned a random set of exactly v bit vector locations, which we call the *item set*.

The two CCBF operations can be implemented by a single server and any number of clients. The protocols are simple and straightforward, save for the calculation of the *tipping point* τ which we present in the next subsection.

In these protocols, the size-s bit vector T is considered *public* or *world-readable*; it is known by all parties at all times. In reality, the server who actually stores T may send it to the client periodically, or whenever a client initiates a Increment or TestCount protocol. However, the bit vector T is only writable by the server.

The $\mathsf{Increment}(x, C)$ protocol, outlined in Algorithm 1, involves the User attempting to set a single bit from 0 to 1 within the item set for x. However, the user is only allowed to write locations within their own user set. So, if there are no 0 bits in the intersection of these two index sets, the user instead changes any other arbitrary 0 bit in its own user set in order to maintain item obliviousness.

Since the bit vector T is considered world-readable, the only communication here is the single index i from client to server over an authenticated channel. In reality, to avoid race conditions, the server will actually send the table entry values $T[i]$ for all $i \in U_C$ to the user first and lock the state of the global bit vector T until receiving the single index response back from the user.

The $\mathsf{TestCount}(x, t)$ protocol is not interactive as it only requires reading the entries of T. The precise computation of the *tipping point* τ is detailed in the next section. Note that this computation depends only on the *total* number of bits set in the bit vector T as well as the parameters s, u, v; therefore the computation of τ is independent of the item x and could for example be performed once by the server and saved without violating item obliviousness.

This protocol is detailed in Algorithm 2.

Algorithm 1 $\mathsf{Increment}(x, C)$

1) User and server separately compute the list of u user locations for user C, $U_C \subseteq \{0, \ldots, s-1\}$.
2) User computes list of v item locations for item x, $V_x \subseteq \{0, \ldots, s-1\}$
3) User checks each location in U_C in the table T to compute a list $S_C = \{i \in U_C \mid T[i] = 0\}$ of *settable* locations for user C
4) If $S_C = \emptyset$, then the user cannot proceed and calls **abort**.
5) Else if $S_C \cap V_x \neq \emptyset$, user picks a uniformly random index $i \leftarrow S_C \cap V_x$ and sends index i to server.
6) Else user picks a random index $i \leftarrow S_C$ and sends index i to server.
7) Server checks that received index i is in the user set U_C and that $T[i] = 0$, then sets $T[i]$ to 1.

Algorithm 2 $\mathsf{TestCount}(x, t)$

1) Use parameters s, u, v and current value of m total number of bits set in T, to compute the tipping point τ.
2) Compute list of v item locations for item x, $V_x \subseteq \{0, \ldots, s-1\}$
3) Check how many bits of T are set for indices in V_x. Return **true** if and only if this count is greater than or equal to τ.

B. Calculating the tipping point

The key to correctness of the TestCount protocol is a calculation of the *tipping point* τ, which is the expected number of 1 bits within any item set, if that item has been incremented t times. We now derive an algorithm to compute this expected value exactly, in $O(tv)$ time and $O(v)$ space.

Let s be the total size of the table T and $m \leq s$ be the total number of calls to Increment so far. That is, m equals the number of 1 bits in T. Recall that $u, v \leq s$ are the number of table entries per user and per item, respectively.

We first derive the probability that two subsets of the s slots have given-size intersection. Next we derive a recursive formula for τ using these intersection probabilities. The nearest integer to τ can then be efficiently computed using a simple dynamic programming strategy.

Intersection probabilities

For the remainder, we use Knuth's notation $n^{\underline{k}}$ to denote the *falling factorial*, defined by

$$n^{\underline{k}} = \frac{n!}{(n-k)!} = n \cdot (n-1) \cdot (n-2) \cdots (n-k+1).$$

Lemma 1. *Let k, a, b, s be non-negative integers with $k \leq b \leq a \leq s$, and suppose S and T are two subsets of a size-s*

set with $|S| = a$ and $|T| = b$, each chosen independently and uniformly over all subsets with those sizes. Then

$$\Pr(|S \cap T| = k) = \frac{a^{\underline{k}} \cdot b^{\underline{k}} \cdot (s-a)^{\underline{b-k}}}{s^{\underline{b}} \cdot k!}. \quad (1)$$

Proof. The number of ways to choose S and T with a size-k intersection, divided by the total number of ways to choose two size-a and size-b sets, equals

$$\frac{\binom{s}{k} \cdot \binom{s-k}{a-k} \cdot \binom{s-a}{b-k}}{\binom{s}{a} \cdot \binom{s}{b}}.$$

This simplifies to (1). ∎

Because the numerator and denominator are each products of $b + k$ single-precision integers, the value of (1) can be computed in $O(b)$ time to full accuracy in machine floating-point precision.

Furthermore, equation (1) has the convenient property that, after altering any value a, b, or k by ± 1, we can update the probability with only $O(1)$ additional computation. So, for example, one can compute the probabilities for every $k \leq b$ in the same total time $O(b)$.

Recurrence for number of unfilled message slots

Fix an arbitrary item x, and let $w \leq v$ denote the number of 0 bits of T within x's item set. Let $k \leq m$ denote the number of Increment operations performed on item x performed so far.

First, for convenience define p_w to be the probability that an arbitrary user is able to write to one of the w remaining unfilled slots for the message. From Lemma 1, we have

$$p_w = 1 - \frac{(s-u)^{\underline{w}}}{s^{\underline{w}}}, \quad (2)$$

which can be computed in $O(w)$ time. In fact, we pre-compute *all* possible values of p_w with $0 \leq w \leq v$ in $O(v)$ total time.

Now consider the random variable for the number of 0 bits within x's item set after k Increment's on x, if the item set originally had w 0 bits. Define $R_{w,k}$ to be the expected value of this random variable, which can be calculated recursively as follows.

If $w = 0$, then the slots are all filled, and if $k = 0$ then there are no more Increment's, so the number of unfilled slots remains at w. Otherwise, the first Increment will fill an additional slot with probability p_w, leaving $w - 1$ remaining unfilled slots, and otherwise will leave w remaining unfilled slots. This implies the following recurrence relation:

$$R_{w,k} = \begin{cases} 0, & w = 0 \\ w, & k = 0 \\ p_w R_{w-1,k-1} + (1-p_w) R_{w,k-1}, & w, k \geq 1 \end{cases}$$

All values of $R_{w,t}$ with $0 \leq w \leq v$ can be computed in $O(tv)$ time and $O(v)$ space, using a straightforward dynamic programming strategy.

Computing the tipping point

We now show how to compute the tipping point value τ, which is the expected number of filled item slots after t Increments on that item, by summing the $R_{w,t}$ values over all possible values of w based on the number of *other* calls to Increment m.

To this end, define q_w to be the probability that $w \leq v$ slots for a given item are unfilled after m total calls to Increment for other items. Because other calls to Increment are for other unrelated items, each one goes to a uniformly-random unfilled slot over the entire size-s table T. Therefore q_w is the same as the probability of a size-m set and a size-v set having intersection size exactly $v - w$. From Lemma 1, this is

$$q_w = \frac{m^{\underline{v-w}} \cdot v^{\underline{v-w}} \cdot (s-m)^{\underline{w}}}{s^{\underline{v}} \cdot (v-w)!}.$$

We can pre-compute all values of q_w for $0 \leq w \leq v$ in total time $O(v)$.

After pre-computing the values of p_w, $R_{w,t}$, and q_w, we can finally express the tipping point τ as a linear combination

$$\tau = v - \sum_{w=0}^{v} q_w R_{w,t}, \quad (3)$$

rounded to the nearest integer.

In total, the computation requires $O(tv)$ time and $O(v)$ space.

C. Tail Bounds

Next, we prove lower and upper bounds on the probability of filling a single additional item slot during an Increment operation, Lemmas 2 and 3 respectively. The proofs, which are intricate but not especially surprising, can be found in Appendix A.

In order to make our scheme practically realizable, we state and prove explicit rather than asymptotic results, with all constants specified. These constants in themselves are not particularly meaningful; rather, they represent the tightest values which worked with our proof techniques and the parameter ranges we deemed reasonable for the application in mind.

Lemma 2. *Let x be an item such that at most τ of x's item slots are filled. If the CCBF parameters s, u, v satisfy $v \geq 7.042652\tau$ and $u \geq 0.5184846\frac{s}{\tau}$, then the probability that a call to* Increment(x, C) *fills in one more of x's item slots is at least* 0.956414.

Lemma 3. *Let x be any item. If the CCBF parameters s, u, v satisfy $371 \leq v \leq 0.00386s$ and $u \leq 3.65151\frac{s}{v}$, then the probability that a call to* Increment(x, C) *fills in one more of x's item slots is at most* 0.974876.

Now we use the probability upper bound to prove an upper bound on the tipping point τ.

Lemma 4. *Let s, u, v be CCBF parameters that satisfy the conditions of Lemma 3, and suppose m, t are integers such that $s \geq 96m$ and $v \leq 7.409t$. Then the tipping point τ, for threshold t and with m total set bits in the table T, is at most* $1.0520553t$.

We can now state our main theorems on the accuracy of the CCBF data structure. Consider a call to the predicate function $\mathsf{TestCount}(x,t)$, which attempts to determine whether the number of prior $\mathsf{Increment}$ calls with the same item x is at least t. Our exact computation of the tipping point $r(t)$ shows that this function always returns the correct answer with at least 50% probability. But of course, so would a random coin flip!

Let k be the *actual* number of calls to $\mathsf{Increment}(x,C)$ that have occurred. Then two kinds of errors can occur: a *false positive* if $\mathsf{TestCount}(x,t)$ returns true but $k < t$, and a *false negative* if $\mathsf{TestCount}(x,t)$ returns false when $k \geq t$. Intuitively, both errors occur with higher likelihood when the true count k is close to t. Our main theorem captures and quantifies this intuition, saying that, ignoring low-order terms, $\mathsf{TestCount}$ is accurate to within a 10% margin of error with high probability.

Theorem 1. *Let n be an upper bound on the total number of calls to* $\mathsf{Increment}$*, and t be a desired threshold for* $\mathsf{TestCount}$*. Suppose the parameters s, u, v for a CCBF data structure satisfy the conditions of Lemma 2, and furthermore that $v \leq 8t$. If the actual number of calls to $\mathsf{Increment}(x,C)$ is at most $t - 2.1\sqrt{\lambda t}$, then the probability $\mathsf{TestCount}(x,t)$ gives a false positive is at most $2^{-\lambda}$.*

Theorem 2. *Let n be an upper bound on the total number of calls to* $\mathsf{Increment}$*, and t be a desired threshold for* $\mathsf{TestCount}$*. Suppose the parameters s, u, v for a CCBF data structure satisfy the conditions of Lemmas 2 and 4. If the actual number of calls to $\mathsf{Increment}(x,C)$ is at least*

$$1.1t + .4\lambda + .7\sqrt{\lambda t}, \tag{4}$$

then the probability $\mathsf{TestCount}(x,t)$ gives a false negative is at most $2^{-\lambda}$.

We can easily summarize the various conditions on the parameters as follows:

Corollary 1. *Let n be a limit on the total number of calls to* $\mathsf{Increment}$*, and t be a desired threshold satisfying $50 \leq t \leq \frac{n}{20}$. Then by setting the parameters of a CCBF according to $s = 96n$, $v = 7.409t$, and $u = 47.31\frac{n}{t}$, any call to $\mathsf{TestCount}(x,t)$ will satisfy the high accuracy assurances of Theorems 1 and 2.*

V. Instantiating FACTS

We are now ready to present our construction of FACTS. This construction is based on the collaborative counting Bloom filter (CCBF) data structure presented in Section IV to obliviously count the number of complaints on each message. It uses an underlying EEMS for sending end-to-end encrypted messages between users.

Setup: The setup procedure for FACTS first sets up the underlying end-to-end encrypted messaging system (EEMS). For simplicity, we assume that there is a fixed number c of users using the system. Setup generates all necessary keys for the server S and all c users and distributes the keys. We note that if the messaging system is already setup, FACTS can simply leverage this for communication. Additionally, the server initializes an empty CCBF data structure.

Sending and receiving messages: We now describe how FACTS originates, forwards, and verifies messages. We start our description with an auxiliary protocol $\mathsf{Originate}(A,x)$ between a user A and the server S to originate a new message x. This protocol is used to create an origination tag tag_x containing information about the message and originator. This tag binds the originator's identity A to the message x to enable recovery upon an audit, while keeping A private from receiving users, and keeping the message x private from the server S.

Roughly, this protocol works by having S produce a signature on (a hash of) the message together with the originator's identity. Due to the use of the hash, S produces this signature without learning anything about the message, while the fact that S includes the originator's identity in this signature prevents a malicious originator from including the wrong identity in the message. Moreover, since the tag is bound to the message, this prevents a replay attack where an adversary reuses tags across messages to change the identity of the originator.

Algorithm 3 $\mathsf{Originate}(A,x)$

1) To originate a message x, the originator A chooses a random salt $r \leftarrow \{0,1\}^\lambda$, computes a salted hash $h = H(r||x)$, and sends h to S.
2) S computes an encryption of the sender's identity, $e \leftarrow \mathsf{Enc}_{PK_S}(A)$, and produces signature $\sigma = \mathsf{Sig}_{SK_S}(h||e)$. S sends the tuple (e,σ) to A.
3) A outputs $\mathsf{tag}_m = (r,e,\sigma)$.

Next, we describe the $\mathsf{SendMsg}$ protocol which makes use of the $\mathsf{Originate}$ protocol to send a message x between clients A and B while preserving (encrypted) information about the originator of x. x can either be a newly originated message or a forward of a previously received message. In either case, $\mathsf{SendMsg}$ runs the $\mathsf{Originate}$ protocol to produce a new tag tag'_x on the message x. In the case of a new message, tag'_x is sent along with the message, while in the case of a forward, it is discarded and the message is forwarded along with its original tag instead.

Algorithm 4 $\mathsf{SendMsg}(A,B,\mathsf{tag}_x,x)$

1) If $\mathsf{tag}_x = \perp$, then x is a new message A wants to originate. A runs $\mathsf{tag}_x \leftarrow \mathsf{Originate}(A,x)$.
2) If $\mathsf{tag}_x \neq \perp$ x is a message that A wants to forward. A runs $\mathsf{tag}'_x \leftarrow \mathsf{Originate}(A,x)$ and discards the output.
3) A sends (tag_x, x) to B using the E2E messaging platform's send protocol.

RcvMsg is a non-interactive algorithm that allows a receiving user to verify the tag, tag_x, affiliated with a message x.

Specifically, the receiver B verifies the server's signature included in tag_x to make sure that the tag indeed corresponds to x and that the originator id has not been modified. Importantly, B can perform this verification without learning the identity of the originator since the tag contains an encryption of this identity (this ciphertext is what is verified by B).

Algorithm 5 $\mathsf{RcvMsg}(A, B, \mathsf{tag}_x, x)$

1) Parse tag_x as $\mathsf{tag}_x = (r, e, \sigma)$
2) Compute $h = H(r||x)$
3) Run $\mathsf{Ver}_{PK_S}(\sigma, (h||e))$ to check that σ is a valid signature by the server on $(h||e)$. If not, then discard the received message.

Complaints and Audit: We now describe how FACTS allows users to complain about received messages and to trigger an audit once enough complaints are registered on a message. For these methods we make extensive use of a CCBF data structure for (approximately) counting complaints and detecting when a threshold of complaints has been reached.

The Complain protocol is used by a receiving user to issue a complaint on a received message (tag_x, x). We assume that prior to issuing a complaint the user verifies that tag_x is valid using the RcvMsg protocol, and thus will only consider the case of valid tags. To issue a complaint on (tag_x, x), the user C calls CCBF.Increment(tag_x, C). As described in Section IV, this runs a protocol with the server in which the user (eventually) sends the location of a bit to flip to 1 to increment the CCBF count for the message x. To prevent malicious adversaries from flooding FACTS with complaints, we enforce a limit of L complaints per user per epoch. Note that since the server knows the identities of complaining users, he can easily enforce this restriction.

Two important observations are in order here. First, we use tag_x rather than the message x as the item to increment in the CCBF. The reason for this is that the tag is unpredictable to an adversary who has not received the message x through FACTS (even if $\mathcal{A}$ knows x). Second, we note that the CCBF.Increment procedure is inherently sequential. It requires that the CCBF table T be locked for the duration of the Increment call to prevent race condition and to maintain obliviousness (see Section IV for discussion). This means that only one user can run this procedure at a time. Thus, we focus on making this procedure as cheap as possible to minimize the impact of this bottleneck. In case multiple clients call Complain at overlapping times, the server can queue these complaints and process them one at a time.

Algorithm 6 $\mathsf{Complain}(C, \mathsf{tag}_x, x)$

1) Parse tag_x as $\mathsf{tag}_x = (r, e, \sigma)$
2) Call CCBF.Increment(C, tag_x)

The Audit protocol checks whether a threshold of complaints has been reached for a given message x and, if so, triggers an audit of this message. This protocol works by using the CCBF.TestCount protocol to check whether the threshold t of complaints has been reached on this message. If this returns True, then the user simply sends (tag_x, x) to the server who first checks the validity of the tag, and then if it's valid, decrypts the corresponding part of the tag to recover the identity of the message originator.

An important observation is that the CCBF.TestCount operation is read-only and thus does not need to block. Thus, unlike the Complain command, many clients can execute the Audit command in parallel.

Algorithm 7 $\mathsf{Audit}(C, \mathsf{tag}_x, x)$

1) Parse tag_x as $\mathsf{tag}_x = (r, e, \sigma)$
2) Call CCBF.TestCount(tag_x, t).
3) If TestCount returns True, x sends (tag_x, x) to S
4) S verifies that the tag is valid by checking the σ is a valid signature on $h||e$ where $h = H(r||x)$.
5) If so, S recovers the identity (A) of the originator by computing $A = \mathsf{Dec}_{SK_S}(e)$.

We note that Audit allows the server to learn the message x and the originator A. We do not specify what the server does upon learning this information, as that is specific to a particular use of FACTS. One possible option is for the server to review x to see if it is truly a malicious message, and if so, block the user A from sending further messages. However, this decision is orthogonal to the FACTS scheme and we do not prescribe a particular action here.

VI. Experimental Evaluations

In this section, we empirically evaluate the accuracy and performance of FACTS. We perform two sets of experiments. The first, measures the error in terms of number of complaints above or below the threshold as a function of the total number of complaints. The second, measures the performance overhead for messaging and complaint as a function of the threshold.

A. Experimental parameters

For our experiments, we set the maximum number of complaints per epoch $n = 1,000,000$. If we consider an epoch of one day, this results in approximately 11.6 complaints per second. To understand accuracy and efficiency of FACTS, we measure them for a range of thresholds $100 \leq t \leq 1000$. With these fixed, we set the remaining parameters according to Corollary 1. In particular, we set the server's storage $s = 96n = 12MB$. The user set size u varies from (approximately) 47,000 to 470,000 bits, while the message set size v goes from (approximately) 740 to 7400.

B. Accuracy and stability

To measure the accuracy of FACTS, we observe the actual number of complaints necessary to cause an audit on a single message as a function of the background noise (i.e., total complaints on other messages). We calculate both the mean and the standard deviation of this value to capture the

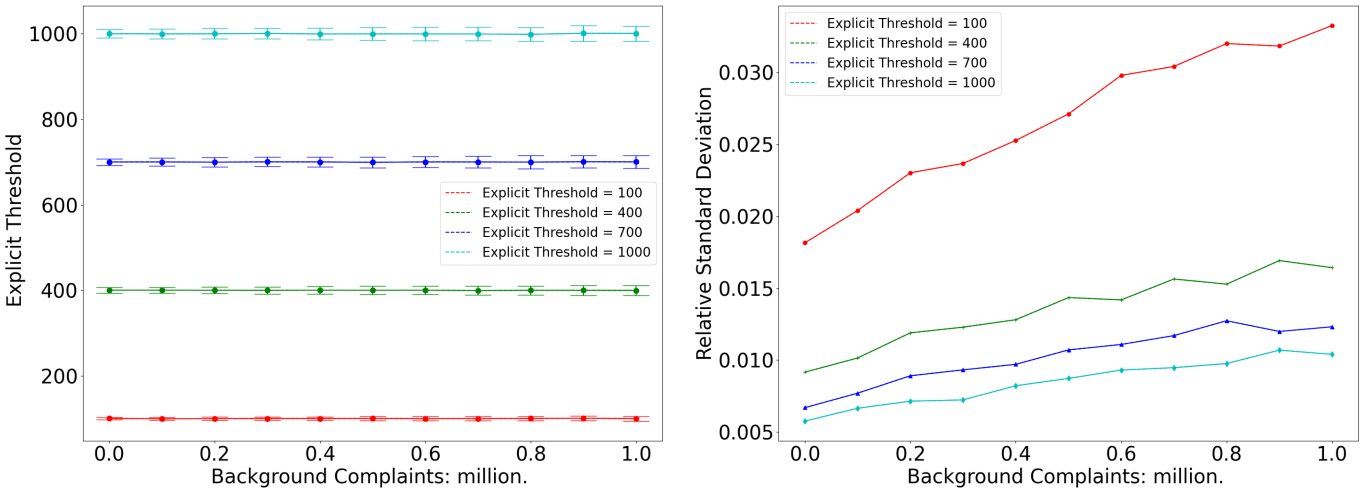

Fig. 1: The (left) Mean and (right) Relative Standard Deviation of Experimental Explicit Threshold vs. The Number of Background Complaints.

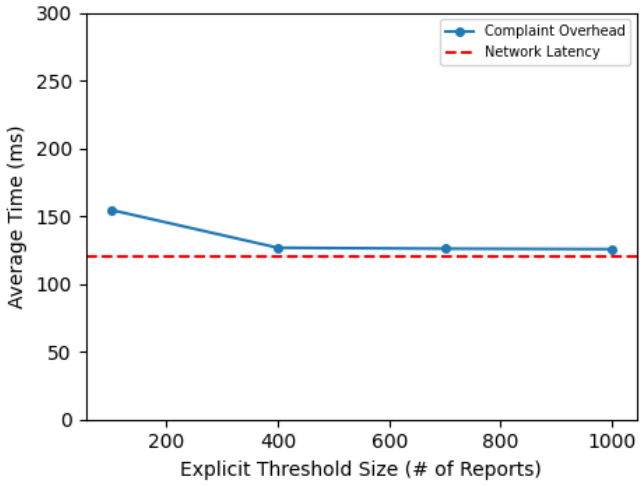

Fig. 2: Average complaint time as a function of the threshold with $n = 1,000,000$ complaints per epoch. Complaint time is measured as the average of 100 samples and has a variance of less than 5ms. Network latency shows the minimum latency required to transmit 3 sequential messages over the network, a lower bound on complaint time.

accuracy and stability of the complaint mechanism. To get a statistically meaningful estimate of these, our experiments run 1000 iterations of each parameter configuration.

The results of our experiments are presented in Figure 1. The left side of this figure shows the mean number of complaints to trigger an audit for a given threshold t. As can be seen from the error bars, the absolute errors in number of complaints is quite small, with a maximum deviation of about 10 complaints at a threshold of 1000. Not surprisingly, we see that this error increases as the background noise increases, but the mean number of complaints remains remarkably steady at the desired value. The right side of Figure 1 shows the relative standard deviation of the number of complaints as a function of background noise. From this graph we can see that the relative error is only a few percent, with a maximum relative error of about 3.5%. Not surprisingly, the threshold 100 measurement incurs the highest relative error because the noise is a much higher ratio when compared to the threshold. These experiments suggest that FACTS achieves good accuracy for a wide variety of threshold and background noise.

C. Performance overhead

Our next set of experiments measures the performance overhead of FACTS as a function of the threshold to start an audit. Specifically, we measure the overhead of sending a message using FACTS, and the cost of issuing a complaint. We note that for the message sending cost, we do not measure the cost of the EEMS communication, instead only measuring the added overhead due to FACTS.

For these experiments, we implemented both the client and server using the Rust programming language. We used SHA-3 for a hash function, and for encryption and signatures we used Rust's ring library [39] implementation of OpenSSL's ChaCha20-Poly1305 protocol and Ed25519 respectively. To instantiate the CCBF, we used a simple library bitvec [35] that allows memory to be bit addressed, rather than byte addressed, which gains us a quick, compact way to store the CCBF data structure.

To simulate network overhead, we implemented a simple web server and client, which communicated over a (simulated) 8 Mbps network with a latency of 80ms, using TLS 1.3. Since we are only measuring the overheads of FACTS over

the underlying EEMS, our measurements did not include the time to send the message over the EEMS, nor the time to establish the TLS connection. All experiments were run on a 4.7Ghz Intel Core i7 with 16GB of RAM, with a sample size of 100 for each metric. As in the accuracy experiments, we set $n = 1,000,000$ and threshold varying from 100 to 1000, with the remaining parameters determined by Corollary 1.

For our measurement of message origination we looked at the cost of originating and sending a message of size 2MB. Creating and sending such a message with the encrypted hash and identity took 98ms, which indicates that the major bottleneck in this process is the 80ms network latency. We see then that when a user wishes to forward a message, they will still call $\mathsf{Originate}(A, x)$, but then forward the original message whereas in an EEMS this would just require a forward. Thus, the overhead of FACTS on a forward is slightly less than 100ms.

Figure 2 shows our measurements of the time to issue a complaint as a function of the audit threshold. The time for this is dominated by the time to retrieve the user set (i.e., the bits that the user can write) from the server. Since the size of this set $u = O(n/t)$, this time grows inversely with the threshold t. Thus, as the threshold increases, the total complaint time decreases very quickly, going down to essentially just the network latency when $t \geq 400$.

These experiments show that both the (added) cost of sending messages and the cost of complaints (for sufficiently large t) are dominated by the networking costs. Thus, as long as the latency of the network is reasonable, FACTS can scale to millions of complaints per day.

VII. Security of FACTS

In this section we analyze the security of FACTS. We provide security definitions capturing the privacy and integrity guarantees provided by FACTS and prove that our protocols described in Section V achieve these definitions.

A. Adversary Model

We consider two different types of adversaries against FACTS. The first is an honest-but-curious server S. Such a server may also collude with some of the users. However, all such users, as well as the server, will follow the protocol. This adversary class models what the FACTS server learns in running the system, so we want to limit what the server learns. However, we have to assume that the server acts honestly, as a malicious server can fully break the integrity and availability of FACTS. For example, since the server produces the signatures binding originators to messages, a malicious adversary with knowledge of this key could arbitrarily assign originators by forging this signature.

We also consider a second type of adversary controlling a group of malicious users who do not collude with the server. Such users may want to violate the confidentiality of FACTS by learning extra information about messages or complaints, beyond what they learn through the messages they validly receive. Or, they may want to break the integrity of the complaint and audit mechanism of FACTS to blame innocent parties for audited messages, or to delay or speed-up the auditing of targeted messages. This models an external adversary, say a malicious company or government, who may want to distribute fake information without being audited or may want to block certain information or users from the system.

B. Privacy

We begin by looking at the privacy guarantees provided by FACTS.

Privacy vs. Server: We first give a definition for privacy against a semi-honest server who may also collude with some semi-honest users. In this setting we aim to argue that unless a message is audited or is received by an adversarial user, the server learns no information about the message or the complaints on the message. In particular, the server should not be able to tell whether any message is a new message or a forward and how many, if any, complaints this message may have. In fact, the only thing that the server learns is the *metadata* of who is sending messages to whom and who is issuing complaints, but not anything more.

Specifically, we propose a real-or-random style definition to capture privacy against the server. This definition captures the fact that the view of the server (and colluding users) until a message is audited or received by a colluding user just consist of random values, and thus is independent of the messages and complaints.

Concretely, we define the following game between an adversary $\mathcal{A}$ controlling the server (and possibly some colluding users) and a challenger.

$\mathsf{Game}^{\mathsf{server\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A})$:

1) The challenger runs $\mathsf{Setup}(c)$ to set up the EEMS with c users. He hands all keys corresponding to corrupted parties to $\mathcal{A}$
2) $\mathcal{A}$ chooses a sequence of messages $((\text{send}, A_0, B_0, \mathsf{tag}_{x_0}, x_0), \ldots, (\text{send}, A_\ell, B_\ell, \mathsf{tag}_{x_\ell}, x_\ell))$[2], and a sequence of complaints $((\text{complain}, C_0, \mathsf{tag}_{x_0^c}, x_0^c), \ldots, (\text{complain}, C_{\ell'}, \mathsf{tag}_{x_{\ell'}^c})$ and interleaves them arbitrarily. We require that none of the sending users (A_i), receiving users (B_i), or complainers (C_i) are controlled by $\mathcal{A}$.
3) The challenger chooses $b \leftarrow \{0, 1\}$ and does the following:
   a) If $b = 0$, Run the $\mathsf{SendMsg}$ and $\mathsf{Complain}$ protocols with inputs supplied by $\mathcal{A}$, giving $\mathcal{A}$ the resulting server view.
   b) If $b = 1$,
      - for each $\mathsf{SendMsg}$ command, choose $r \leftarrow \{0,1\}^\lambda$ and send this to S. Choose $x' \leftarrow \{0,1\}^{|x|+|\mathsf{tag}_x|}$ and send x' from A_i to B_i using EEMS.send.

[2]We note that since $S \in \mathcal{A}$, $\mathcal{A}$ can produce valid-looking tags for each of these messages by producing the necessary signatures.

- The challenger maintains a set USED $\subseteq [s]$[3]. For each Complain command, the challenger chooses $ind \leftarrow [s] \setminus$ USED, sends ind from u_i to S, and adds ind to USED.

4) $\mathcal{A}$ outputs a bit b'
5) We say that $\mathcal{A}$ has advantage

$$\mathsf{Adv}^{\mathsf{server\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A}) = |\Pr[b = b'] - 1/2|.$$

Definition 1 (Privacy vs. Server)**.** *A FACTS scheme is* private against a semi-honest server *if the adversary has a negligible advantage in the game above* $\mathsf{Adv}^{\mathsf{server\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A}) \leq \mathsf{negl}(\lambda)$

Theorem 3. *FACTS is private against a semi-honest server*

Proof sketch. First, consider the server's view on a SendMsg command. This view consists of a message $h = H(r||m)$ for $r \leftarrow \{0,1\}^\lambda$ and the leakage from EEMS.send, i.e., the identities A and B, as well as $|(\mathsf{tag}_x, x)|$. Since the challenger uses the same sender, receiver, and message length, the only thing left to prove is that h is indistinguishable from random. Since r is chosen uniformly at random, and H is a random oracle, $H(r||m)$ is uniformly random to $\mathcal{A}$ unless $\mathcal{A}$ queries $H(r||m)$. However, since $\mathcal{A}$ makes at most $\mathsf{poly}(\lambda)$ queries to H, the probability that he makes this query is at most $\mathsf{poly}(\lambda)/2^\lambda \leq \mathsf{negl}(\lambda)$.

Next, we consider the Complain commands. The server's view on a complaint consists of the complainer's ID C and an index in the CCBF to flip to 1. In a real execution of Complain, this index is chosen at random from the set $S_C \cap V_x$ where $S_C = \{i \in U_C \mid T[i] = 0\}$ and V_x is the list of item locations for x.[4] However, since U_C and V_x are chosen at random, we can equivalently sample a random 0-index in the bit vector T and then choose U_C and V_x conditioned on them containing this location. Hence the location sent to the server is uniformly random unless $\mathcal{A}$ makes the corresponding H query, which only happens with $\mathsf{negl}(\lambda)$ probability. ∎

The above theorem states that, beyond the meta-data of who sent a message to whom and who has sent complaints and when, FACTS reveals no information about messages and complaints to a semi-honest server until an audit occurs (or a malicious user receives a message). Moreover, the view of the server is completely random when conditioned on the meta-data. Now, suppose that a message x is audited (or is received by an adversary-controlled user). When this happens, the adversary learns the tag and message (tag_x, x). This enables $\mathcal{A}$ to learn the identity of the originator (by decrypting it from tag_x) and to learn the entire history of this message, i.e., the transmission and complaint history of x. However, since the server's view of all other messages is indistinguishable from independent random strings (modulo the meta-data), the adversary does not learn anything more about these messages as a result of an audit on x.

[3]Recall that s is the size of the CCBF bit vector T

[4]Technically, the item used in the CCBF is the tag tag_x, but we use x here for ease of notation.

Privacy vs. Users We now proceed to analyze security of our protocol against (possibly malicious) users that are not colluding with the server. This models the case of a third party adversary that tries to learn information about the messages and complaints in FACTS. Here, we no longer assume that a message x is never received by a malicious user and thus we cannot use a real-or-random style definition as before. Instead, we argue that a user cannot distinguish a new message from a forwarded message unless another corrupted user has previously seen that message. This also shows that a malicious user cannot learn the identity of the message originator. Since users do not receive any communication on complaints, we only consider message privacy here.

Concretely, we define the following game between an adversary $\mathcal{A}$ controlling a set of users, and a challenger.

$\mathsf{Game}^{\mathsf{user\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A})$:

1) The challenger runs $\mathsf{Setup}(c)$ to set up the EEMS with c users and gives all key material for the corrupted users to $\mathcal{A}$. Let $B \in \mathcal{A}$ be a user controlled by the adversary.
2) $\mathcal{A}$ chooses messages x, x' s.t. $|x| = |x'|$ and honest users $O, A \notin \mathcal{A}$
3) The challenger chooses $b \mid \{0,1\}$ and does the following:
   a) If $b = 0$, the challenger runs $\mathsf{SendMsg}(O, A, \perp, x')$ and $\mathsf{SendMsg}(A, B, \perp, x)$ with $\mathcal{A}$ receiving the view of B.
   b) If $b = 1$, the challenger runs $\mathsf{SendMsg}(O, A, \perp, x)$ and $\mathsf{SendMsg}(A, B, \mathsf{tag}_x, x)$ (where tag_x is the tag received by A from O).
4) $\mathcal{A}$ outputs a bit b'
5) We say that $\mathcal{A}$ has advantage

$$\mathsf{Adv}^{\mathsf{user\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A}) = |\Pr[b = b'] - 1/2|.$$

Definition 2 (User privacy)**.** *A FACTS scheme achieves* privacy against malicious users *if the adversary has a negligible advantage in the game above* $\mathsf{Adv}^{\mathsf{user\text{-}privacy}}_{\mathsf{EEMS}}(\mathcal{A}) \leq \mathsf{negl}(\kappa)$

Theorem 4. *FACTS achieves privacy against malicious users.*

Proof sketch. The view of B on an execution of $\mathsf{SendMsg}(\cdot, B, \mathsf{tag}_x, x)$ consists of the received message and tag (tag_x, x) where $\mathsf{tag}_x = (r, e, \sigma)$. Since e is a semantically secure encryption of the identity of the originator, $\mathcal{A}$ cannot distinguish between the case when $e = \mathsf{Enc}(A)$ (when $b = 0$) and the case when $e = \mathsf{Enc}(O)$ (when $b = 1$) except with advantage negligible in κ. Additionally, since tag_x is generated identically both when $b = 0$ and $b = 1$ except for this change in e, this means that tag_x does not help $\mathcal{A}$ distinguish between these two cases. ∎

C. Integrity

We now turn to the integrity guarantees provided by FACTS. We aim for a few different notions of integrity to show that malicious users cannot interfere with the complaint and audit process. First, no adversary controlling a subset of the users should be able to frame an honest user as the originator

of an audited message he did not originate. Second, an adversary controlling a subset of the users should not be able to significantly delay the audit of a malicious message. In particular, such an adversary should not be able to prevent a malicious message sent by one of his users from being audited. Finally, an adversary controlling a small set of users should not be able to significantly speed up the auditing of a targeted message. In particular, such an adversary should not be able to cause an audit without complaints from some honest users.

We begin by defining the following game between a challenger and an adversary $\mathcal{A}$ controlling a subset of the users to capture the inability of an adversary to forge a valid tag that it has not seen before.

$\mathsf{Game}^{\mathsf{unforgeability}}_{\mathsf{EEMS}}(\mathcal{A})$:

1) The challenger runs $\mathsf{Setup}(c)$ to set up the EEMS with c users and gives all key material for the corrupted users to $\mathcal{A}$.
2) $\mathcal{A}$ requests $\mathsf{SendMsg}$ operations on messages of its choice both from honest and corrupted users. ($\mathcal{A}$ is given the view of corrupted users in all these executions consisting of (tag_x, x).)
3) $\mathcal{A}$ outputs a tag, message pair (tag_y, y)
4) We say that $\mathcal{A}$ WINS if tag_y is a valid tag for message y with originator $O \notin \mathcal{A}$, and there has not been a prior command $\mathsf{SendMsg}(O, \cdot, \perp, y)$.

Definition 3 (No framing)**.** *We say that a FACTS scheme disallows* framing *if for any PPT $\mathcal{A}$, $\mathcal{A}$ WINS in the above game with probability at most* $\mathsf{negl}(\kappa)$.

Theorem 5. *The FACTS scheme is unforgeable.*

Proof sketch. A valid tag tag_y with originator O consists of $\mathsf{tag}_y = (r, e, \sigma)$ where r is a random seed s.t. $H(r||y) = h$, $e = \mathsf{Enc}_{PK_S}(O)$, and $\sigma = \mathsf{Sig}_{SK_S}(h||e)$. Thus, to frame O, $\mathcal{A}$ needs to produce a valid signature on $h||\,\mathsf{Enc}(O)$. $\mathcal{A}$ can observe tags from polynomially many messages originated by $\emptyset$, but except with probability negligible in λ none of them will have the same value h. Thus, by the unforgeability of Sig, $\mathcal{A}$ cannot produce the necessary signature except with probability negligible in κ. ∎

Next, we give a definition that captures the ability of an adversary controlling a subset of the users to delay the audit of a particular message. Our goal is to show that the adversary cannot protect a malicious message from being audited.

Specifically, we define the following game,

$\mathsf{Game}^{\mathsf{no-delay}}_{\mathsf{EEMS}}(\mathcal{A})$:

1) The challenger runs Setup to set up the EEMS with c users and gives all key material for the corrupted users to $\mathcal{A}$.
2) $\mathcal{A}$ issues a single $\mathsf{SendMsg}(A, B, x)$ command with $A \in \mathcal{A}$ to produce tag_x
3) $\mathcal{A}$ outputs a list of $\mathsf{Complain}$ commands with at most n total complaints, of which at least ℓ are complaints on tag_x.
4) The challenger runs the specified complaint commands, and then runs $\mathsf{Audit}(A, \mathsf{tag}_x, x)$
5) We say that $\mathcal{A}$ WINS if this audit is not successful (i.e., the audit threshold is not reached).

Definition 4 (No delay)**.** *We say that a FACTS scheme is* ℓ-audit delay resilient *for integer $\ell < n$ if for any PPT $\mathcal{A}$, $\mathcal{A}$ WINS in the above game with probability at most* $\mathsf{negl}(\lambda)$.

Theorem 6. *The FACTS scheme is ℓ-audit delay resilient for any $\ell \geq 1.1t + .4\lambda + .7\sqrt{\lambda t}$.*

Proof sketch. This follows immediately from Theorem 2 ∎

Next, we define the following game to capture the ability of a small number of malicious users to cause the audit of some message. Importantly, this definition also captures the case where malicious users try to audit an honest message (on which there are no complaints by honest users). Specifically, the following game is between an adversary $\mathcal{A}$ corrupting at most ℓ users and a challenger

$\mathsf{Game}^{\mathsf{no-speedup}}_{\mathsf{EEMS}}(\mathcal{A})$:

1) The challenger runs Setup to set up the EEMS with c users and gives all key material for the ℓ corrupted users to $\mathcal{A}$.
2) The challenger runs a single $\mathsf{SendMsg}(A, B, x)$ command for $A \notin \mathcal{A}$ and $B \in \mathcal{A}$.
3) $\mathcal{A}$ may issue at most L $\mathsf{Complain}$ commands per each user he controls.[5]
4) The challenger runs the specified $\mathsf{Complain}$ commands, and then runs $\mathsf{Audit}(\cdot, \mathsf{tag}_x, x)$.
5) We say that $\mathcal{A}$ WINS if this audit is successful.

Definition 5 (No speed up)**.** *We say that a FACTS scheme is* ℓ-party audit speed-up resilient *if for any PPT $\mathcal{A}$ controlling at most ℓ users, $\mathcal{A}$ WINS in the above game with probability at most* $\mathsf{negl}(\lambda)$.

Theorem 7. *The FACTS scheme is ℓ-party audit speed-up resilient for $\ell \leq (t - 2.1\sqrt{\lambda t})/L$.*

Proof sketch. This follows immediately from Theorem 1 because each user $\in \mathcal{A}$ makes at most L complaints. ∎

VIII. Alternative FACTS

In this section we describe several optimizations or enhancements to the basic FACTS protocol.

Revealing even less during audits. Recall that the FACTS system we presented reveals two things to the server (or an auditor) after the threshold of complaints has been reached: the user id of the message's originator, and the contents of the message itself. Indeed, one of our motivations was to avoid revealing the entire path or tree of message forwarding as in prior work [43].

However, in some environments, even this may be too much to release to a service provider that could be, for instance, compromised or influenced by an oppressive regime. An advantage of FACTS is that the system for tallying complaints actually does not *require* this information in order to function

[5] Recall that FACTS enforces a limit of L complaints per user per epoch.

properly. Here we briefly sketch simple modifications to the scheme to achieve this additional hiding, with a note of caution that we have not analyzed the formal security under these variants.

Hiding the originator's identity entails omitting the encrypted user id from the origination protocol. To do this, the server's signature σ of the message hash and sender identity should be replaced with a *blind signature* of the message hash only. In this way, a later audit which reveals the (unblinded) signature will not reveal anything about the originator's identity. The disadvantage of course would be that there is no way for the system to identify and penalize users who regularly submit fake news to the platform.

To hide the message contents, these would simply be omitted from what is sent to the auditor once the threshold is reached. In this case, the notion of "audit" may be understood to be simply confirming that *some* message (with the given hash) has passed the threshold of complaints, and publishing the hash to all users as *potentially* fake news that has received a large number of complaints. The client software could easily be configured to flag such messages as they are received afterwards, without ever revealing to the server any contents or recipients of such message. Here the disadvantage is obviously that no third-party auditing or fact-checking is possible, raising the possibility of false positives in which messages are flagged.

Throttling complaints. The FACTS system and underlying CCBF data structure assume a global limit n on the number of complaints per epoch, but do not require any per-user limit besides the natural limit of u, the size of the user set.

However, there is some potential for abuse by users who issue many complaints in a single epoch: they may attempt to "attack" another known message by issuing multiple complaints that set bits in that message's user set; they may collude with others and attempt to go over the total per-epoch limit of n complaints; or they may simply attempt a denial-of-service attack to prevent other complaints from being issued.

An simple solution to these problems is to apply a limit $\ll u$ on the maximum number of complaints per user per epoch. This is easy for the server to apply, since users are authenticated during the Complain protocol. More nuanced limits based on a user's reputation or longevity on the platform could also be applied.

Users with a small "quota" of allowed complaints per epoch could even be encouraged to participate initially in the complaint process by forwarding questionable content to a trusted reputable user on the system, who would then presumably apply their own judgment and possible issue a complaint in turn. This idea is aligned with many existing content moderation settings on (unencrypted) social media platforms.

Handling epoch rollover. As described, the FACTS system resets all counters at the end of a single epoch. However, this may mean that if a "fake news" message is first detected towards the end of an epoch, the complaints for this message may get split between the current and next epochs and thus fail to trigger an audit in either epoch.

A potential solution to this problem is to always run two epochs concurrently, where each epoch lasts for time t, and the epochs start times are $t/2$ apart. Users complain in both of the epochs, and an audit occurs if the number of complaints in either epoch exceed the threshold. This way, regardless when a "fake news" message is first detected, there will be an epoch with at least $t/2$ time left to accumulate complaints. Since we assume that fake news messages are ones that are received and complained on by many users, and that users are likely to complain shortly after first receiving a message, this provides enough time for a threshold of complaints to be reached.

Regional complaint servers. The most significant performance bottleneck in FACTS is the necessary global lock on the table T while a single user is waiting to download their user set U_C and reply with their complaint index. Even though the communication size is quite small for practical settings, the inherent latency across global communications networks may impose a challenge.

For example, if many complaining users have a round-trip latency of more than 200ms, then the global complaint rate among all users cannot be higher than 5 complaints per second, or some 432,000 complaints per day, regardless of any parameter settings or chosen epoch length.

One possible solution for a large-scale platform facing this issue would be to allow multiple local complaint servers, each with their own CCBF table T, to independently operate and accumulate complaints per messages. This makes sense, as most targeted misinformation content is local to a given country or region, and it would still be possible for each regional server to share audited message information with others in order to prevent spread of viral false content between regions.

Third-party audits. While many messaging and social media platforms currently employ their own "in-house" teams for content moderation, there have been some attempts at separating the role of the server from that of auditor.

From a protocol standpoint, we can imagine a separate Server and Auditor: the former is semi-honest, handles the encrypted messaging system and maintains the public CCBF table T. The Auditor is fully honest and non-colluding, but computationally limited; intuitively, the third-party Auditor should only be involved once a messaged has passed the desired threshold of complaints.

The FACTS system supports this option easily with the need for any additional cryptographic setup during origination. Because the CCBF table T is globally shared among all users as well as the Auditor, any complaining user who computes TestCount on their own to see that the probabilistic threshold has been surpassed, can then forward their complaint (i.e., the opened message) directly to the Auditor. Being fully honest, the Auditor may hold a copy of the decryption key from origination and use this to determine what kind of action

may be necessary (such as suspending the originating user's account, flagging the message, etc.).

While it doesn't appear idea imposes any additional interesting challenges from a cryptographic standpoint, it could be useful for some kinds of messaging platforms.

Hiding message metadata. Our FACTS system is certainly no more private than the underlying EEMS which is being used to actually pass messages between users. In our analysis, we explicitly assumed that the EEMS leaks metadata on the sender and recipient of each message, but not the contents.

However, some existing EEMS attempt to also obscure this metadata in transmitting messages, so that the server does not learn both sender and recipient of any message. This can trivially be accomplished by foregoing a central server and doing peer-to-peer communication (note that FACTS may still be useful as a central complaint repository); or using more sophisticated cryptography to hide metadata [6], [10], [41].

Of particular interest for us is the recently deployed *sealed sender* mechanism on the popular Signal platform [29]. The goal in this case is to obscure the sender, but not the recipient, from the server handling the actual message transmission. We note that this concept plays particularly well with FACTS, as the additional leakage in our protocol of the identity of each complaining user, can be presumably correlated via timings with the receipt of some message, but this is exactly what is revealed under sealed sender already! Both systems thus work to still hide message sender and originator identities (at least until an audit is performed).

However, note that recent work [31] has shown that some timing attacks are still possible under sealed sender, and the same attacks would apply just as well to FACTS. But the solutions proposed in [31] might also be deployed alongside FACTS to prevent such leakage; we leave the investigation of this question for future work.

IX. Related Work

Message Franking: The most common approach today for reporting malicious messages in encrypted messaging systems is *message franking* [11], [21], [42]. Message franking allows a recipient to prove the identity of the sender of a malicious message. However, message franking is focused on identifying the last sender of a message, whereas we are interested in identifying the originator. Moreover, message franking does not provide any threshold-type guarantees to prevent unmasking of senders given only one (or a few) complaints.

Oblivious RAM (ORAM): Oblivious Random-Access Memory (ORAM) [18], [20], [34] allows a client to obliviously access encrypted memory stored on a server without leaking the access pattern to the server. The standard ORAM definition assumes a single user with full control over the database. While some important progress has been made on multi-client ORAM protocols [7], [22], [30], these solutions are still not scalable to millions of malicious users as would be needed for our application.

Oblivious Counters and Oblivious Data-Structures: Like CCBF, oblivious counters [17], [26] build counters that can be stored and incremented without revealing the value of the counter. However, these techniques focus on exact counting, and do not provide efficient ways for storing large numbers of counters, as needed for our applications. More generally, oblivious data-structures, e.g. [27], [38], [45] construct higher-level data structures such as heaps, trees, etc. to enable oblivious operations over encrypted data. However, these largely focus on higher-level applications and do not provide the compression achieved by CCBF.

Privacy-Preserving Sketching: CCBF can be viewed as a small data structure (a *sketch*) for storing the counts of complaints on a large set of messages. There has indeed been a lot of recent interest (e.g., [3], [8], [12], [14], [25], [32], [44]) in private sketching algorithms for cardinality estimation, frequency measurement, and other approximations. However, these works generally focus on a multi-party setting, with multiple parties running secure computation to evaluate the statistic in question. Since our goal was to restrict ourselves to user-server communication only, such techniques do not seem applicable to our setting.

Acknowledgements

Daniel S. Roche is supported by ONR grant N0001420WX01412; Arkady Yerukhimovich is supported by NSF grant CNS-1955620, and by a Facebook Research Award.

We thank Adam Aviv for introducing us to this problem and for many insightful discussions. We also thank the anonymous reviewers and Hamed Okhravi for helping us significantly improve the presentation of our results.

References

[1] How does YouTube combat misinformation? https://www.youtube.com/howyoutubeworks/our-commitments/fighting-misinformation/#policies.

[2] How is Facebook addressing false information through independent fact-checkers? https://www.facebook.com/help/1952307158131536.

[3] Vikas G. Ashok and Ravi Mukkamala. A scalable and efficient privacy preserving global itemset support approximation using bloom filters. In *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*, pages 382–389, 2014.

[4] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

[5] Russell Brandom. Apple says collision in child-abuse hashing system is not a concern. *The Verge*, August 2021.

[6] J Brooks et al. Ricochet: Anonymous instant messaging for real privacy, 2016.

[7] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client ORAM. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[8] Seung Geol Choi, Dana Dachman-Soled, Mukul Kulkarni, and Arkady Yerukhimovich. Differentially-private multi-party sketching for large-scale statistics. *PoPETs*, 2020(3):153–174, July 2020.

[9] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[10] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 147–162, 2013.

[11] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. Fast message franking: From invisible salamanders to encryptment. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 155–186, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[12] Rolf Egert, Marc Fischlin, David Gens, Sven Jacob, Matthias Senker, and Jörn Tillmanns. Privately computing set-union and set-intersection cardinality via bloom filters. In *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings*, pages 413–430, 2015.

[13] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[14] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. Distributed measurement with private set-union cardinality. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2295–2312, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[15] R.Stuart Geiger. Bot-based collective blocklists in twitter: the counterpublic moderation of harassment in a networked public space. *Information, Communication & Society*, 19:787–803, 06 2016.

[16] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178, Bethesda, MD, USA, May 31 – June 2, 2009. ACM Press.

[17] Eu-Jin Goh and Philippe Golle. Event driven private counters. In Andrew Patrick and Moti Yung, editors, *FC 2005*, volume 3570 of *LNCS*, pages 313–327, Roseau, The Commonwealth Of Dominica, February 28 – March 3, 2005. Springer, Heidelberg, Germany.

[18] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194, New York City, NY, USA, May 25–27, 1987. ACM Press.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

[20] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[21] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. Message franking via committing authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 66–97, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[22] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A. Yavuz. MOSE: Practical multi-user oblivious storage via secure enclaves. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, page 17–28, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Mike Isaac and Kevin Roose. Disinformation spreads on whatsapp ahead of brazilian election. https://www.nytimes.com/2018/10/19/technology/whatsapp-brazil-presidential-election.html, 2018.

[24] Rawane Issa, Nicolas AlHaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender. Cryptology ePrint Archive, 2021.

[25] Rob Jansen and Aaron Johnson. Safely measuring tor. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1553–1567, Vienna, Austria, October 24–28, 2016. ACM Press.

[26] Jonathan Katz, Steven Myers, and Rafail Ostrovsky. Cryptographic counters and applications to electronic voting. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 78–92, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

[27] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525, Kaoshiung, Taiwan, R.O.C., December 7–11, 2014. Springer, Heidelberg, Germany.

[28] Anunay Kulshrestha and Jonathan Mayer. Identifying harmful media in end-to-end encrypted communication: Efficient private membership computation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 893–910. USENIX Association, August 2021.

[29] Joshua Lund. Technology preview: Sealed sender for signal, Oct 2018.

[30] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client oram. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security*, pages 645–664, Cham, 2017. Springer International Publishing.

[31] Ian Martiny, Gabriel Kaptchuk, Adam Aviv, Dan Roche, and Eric Wustrow. Improving signal's sealed sender. 01 2021.

[32] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *NDSS 2016*, San Diego, CA, USA, February 21–24, 2016. The Internet Society.

[33] Michael Mitzenmacher. Compressed bloom filters. In Ajay D. Kshemkalyani and Nir Shavit, editors, *20th ACM PODC*, pages 144–150, Newport, Rhode Island, USA, August 26–29, 2001. ACM.

[34] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[35] Alexander Payne. bitvec. https://lib.rs/crates/bitvec, 11 2021.

[36] Charlotte Peale, Saba Eskandarian, and Dan Boneh. Secure source-tracking for encrypted messaging. In *ACM CCS 21: 28th Conference on Computer and Communications Security*, Virtual Conference, Korea, November 15–19, 2021. ACM Press.

[37] Elyse Samuels. How misinformation on whatsapp led to a mob killing in india. https://www-washingtonpost-com.proxygw.wrlc.org/politics/2020/02/21/how-misinformation-whatsapp-led-deathly-mob-lynching-india/ics/2020/02/21/how-misinformation-whatsapp-led-deathly-mob-lynching-india/, 2020.

[38] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy*, pages 842–858, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

[39] Brian Smith. ring. https://lib.rs/crates/ring, 11 2021.

[40] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[41] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440. ACM, 2017.

[42] Nirvan Tyagi, Paul Grubbs, Julia Len, Ian Miers, and Thomas Ristenpart. Asymmetric message franking: Content moderation for metadata-private end-to-end encryption. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 222–250, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[43] Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. Traceback for end-to-end encrypted messaging. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 413–430. ACM Press, November 11–15, 2019.

[44] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S. Dov Gordon. Stormy: Statistics in tor by measuring securely. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 615–632. ACM Press, November 11–15, 2019.

[45] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 215–226, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[46] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

APPENDIX

We now complete the proofs of lemmas and theorems in Section IV-C.

We start with the following standard way to approximate numbers near 1 with exponentials.

Lemma 5. *For any real constant $\alpha > 0$, and any real x with $0 < x \leq \alpha$, we have*

$$\exp\left(-\tfrac{1}{\alpha}\ln\tfrac{1}{1-\alpha}\cdot x\right) \leq 1 - x < \exp(-x).$$

We also re-state this straightforward consequence of the Hoeffding/Chernoff bound on the sum of random variables:

Lemma 6. *Let $X_1, \ldots, X_n$ be independent Poisson trials, and write $Y = \sum_i X_i$ for their sum. If $\mathbb{E}[Y] = \mu$, then for any $\delta > 0$, each of $\Pr(Y \geq \mu + \delta)$ and $\Pr(Y \leq \mu - \delta)$ are at most $\exp(-2\delta^2/n)$.*

We now recall and prove the building-block lemmas from Section IV-C.

Lemma 2. *Let x be an item such that at most τ of x's item slots are filled. If the CCBF parameters s, u, v satisfy $v \geq 7.042652\tau$ and $u \geq 0.5184846\frac{s}{\tau}$, then the probability that a call to $\mathsf{Increment}(x, C)$ fills in one more of x's item slots is at least 0.956414.*

Proof. From (2), we know this probability is exactly $p_w = 1 - \frac{(s-u)^{\underline{w}}}{s^{\underline{w}}}$, where $w = v - \tau$ is the number of unfilled slots remaining. Using Lemma 5 we have $\frac{(s-u)^{\underline{w}}}{s^{\underline{w}}} \leq \left(1 - \frac{u}{s}\right)^w \leq \exp(-uw/s)$, which means that

$$p_w \geq 1 - \exp\left(-\tfrac{u(v-\tau)}{s}\right) = 1 - \exp\left(-\tfrac{u\tau}{s}\cdot\left(\tfrac{v}{\tau} - 1\right)\right).$$

Applying the two lower bounds on $\frac{u\tau}{s}$ and $\frac{v}{\tau}$ from the lemma statement yields the claimed result. ∎

Lemma 3. *Let x be any item. If the CCBF parameters s, u, v satisfy $371 \leq v \leq 0.00386s$ and $u \leq 3.65151\frac{s}{v}$, then the probability that a call to $\mathsf{Increment}(x, C)$ fills in one more of x's item slots is at most 0.974876.*

Proof. Using again (2), the probability is exactly $p_w = 1 - \frac{(s-u)^{\underline{w}}}{s^{\underline{w}}}$, where again $w \leq v$ is the number of unfilled slots for item x. Then

$$\frac{(s-u)^{\underline{w}}}{s^{\underline{w}}} \geq \frac{(s-u)^{\underline{v}}}{s^{\underline{v}}} \geq \left(\frac{s-u-v+1}{s-v+1}\right)^v > \left(1 - \frac{u}{s-v}\right)^v.$$

Using upper bounds on $\frac{v}{s}$ and u from the lemma statement, we have

$$p_w < 1 - \left(1 - \frac{u}{s-v}\right)^v \leq 1 - \left(1 - 3.66567\tfrac{1}{v}\right)^v.$$

Finally, the lower bound on v from the lemma statement shows $3.66567/v \leq 0.00989$, and so we can finally use the lower exponential bound of Lemma 5 to obtain the stated result. ∎

Lemma 4. *Let s, u, v be CCBF parameters that satisfy the conditions of Lemma 3, and suppose m, t are integers such that $s \geq 96m$ and $v \leq 7.409t$. Then the tipping point τ, for threshold t and with m total set bits in the table T, is at most $1.0520553t$.*

Proof. The tipping point τ is the expected number of slots filled in the table if t of the m total calls to Increment were actually called on this particular item.

We can divide the calls to Increment into two groups: the t calls for item x, and the $m - t$ calls for other items. The expected number of slots within x's item set filled by the first group is at most $0.974876t$, from Lemma 3.

For the second group, these calls to Increment on unrelated items are distributed uniformly at random among all table indices, and so their expected fraction within this item set is the same as their overall fraction in the table. Therefore, the expected number of slots filled by calls to Increment on other items is at most

$$\frac{(m-t)v}{s} < \frac{mv}{s} \leq \frac{7.409}{96}t.$$

By linearity of expectation, we can sum these two to obtain an upper bound on the total expected tipping point as given in the lemma statement. ∎

Now we can proceed to the proofs of the main theorems on the accuracy of the CCBF.

Theorem 1. *Let n be an upper bound on the total number of calls to* Increment*, and t be a desired threshold for* TestCount*. Suppose the parameters s, u, v for a CCBF data structure satisfy the conditions of Lemma 2, and furthermore that $v \leq 8t$. If the actual number of calls to $\mathsf{Increment}(x, C)$ is at most $t - 2.1\sqrt{\lambda t}$, then the probability $\mathsf{TestCount}(x, t)$ gives a false positive is at most $2^{-\lambda}$.*

Proof. Let τ_t be the tipping point for any actual number $m \leq n$ of total set bits in the table T and for the given threshold t. And consider random variables $X_1, \ldots, X_v$ for the v slots assigned to item x, where each X_1 is 0 or 1 depending on whether the corresponding slot in table T is 0 or 1. We want to know the probability that the sum of the X_i's is at least τ_t, which is what would cause $\mathsf{TestCount}(x, t)$ to produce a false positive.

Let $k = t - 2.1\sqrt{\lambda t}$ be the actual number of calls to Increment on item x, and write τ_k for the tipping point at threshold k. By definition and the exact calculations for τ_k outlined earlier, we know that $\mathbb{E}[\sum X_i] = \tau_k$.

The difference between these two tipping points $\tau_t - \tau_k$ is the expected number of extra slots filled by $t - k$ calls to Increment, which from Lemma 2 is at least

$$0.956414(t-k) = 0.956414 \cdot 2.1\sqrt{\lambda t} \geq \sqrt{\tfrac{\lambda v}{2}},$$

where in the last step we used the upper bound on v from the assumptions of the theorem.

The variables X_i are not independent, but they are *negatively correlated*, meaning that the whenever one slot is filled, it only decreases the likelihood that another is filled; intuitively, this is because there are now fewer chances to fill the other slot. Therefore we can apply the Hoeffding bound in this direction (Lemma 6) to say that

$$Pr\left(\sum X_i \geq \tau_k + \sqrt{\tfrac{\lambda v}{2}}\right) \leq \exp(-\lambda),$$

as required. ∎

Theorem 2. *Let n be an upper bound on the total number of calls to* Increment*, and t be a desired threshold for* TestCount*. Suppose the parameters s, u, v for a CCBF data structure satisfy the conditions of Lemmas 2 and 4. If the actual number of calls to* Increment(x, C) *is at least*

$$1.1t + .4\lambda + .7\sqrt{\lambda t}, \tag{4}$$

then the probability TestCount(x, t) *gives a false negative is at most $2^{-\lambda}$.*

Proof. Writing k for the actual number of complaints given in (4), we need a tail bound on the probability that, after k calls to Increment on the same item x, there are still fewer than $1.0520553t$ slots of x's item set filled in, where the latter constant comes from applying the upper bound on the tipping point from Lemma 4.

For this, we need a lower bound on the *expected* number of bits set after k calls to Increment on item x; from Lemma 2 this is at least $0.956414k$.

Now we can apply the Hoeffding bound (Lemma 6), with $\mu = 0.956414k$ and $\mu + \delta = 1.0520553t \geq \tau$ to see that the probability that less than τ bits of x's user set are flipped is at most

$$\begin{aligned}
&\exp\left(-2(1.0520553t - 0.956414k)^2/k\right) \\
\leq &\exp\left(-2\left(0.38\lambda + 0.66\sqrt{\lambda t}\right)^2/k\right) \\
\leq &\exp\left(-\frac{.28\lambda^2 + \lambda\sqrt{\lambda t} + .87\lambda t}{.4\lambda + .7\sqrt{\lambda t} + 1.1t}\right) \\
\leq &\exp(-.7\lambda) \leq 2^{-\lambda}.
\end{aligned}$$

■