Traceback for End-to-End Encrypted Messaging

Nirvan Tyagi
Cornell University

Ian Miers
Cornell Tech
University of Maryland

Thomas Ristenpart
Cornell Tech

ABSTRACT

Messaging systems are used to spread misinformation and other malicious content, often with dire consequences. End-to-end encryption improves privacy but hinders content-based moderation and, in particular, obfuscates the original source of malicious content. We introduce the idea of message traceback, a new cryptographic approach that enables platforms to simultaneously provide end-to-end encryption while also being able to track down the source of malicious content reported by users. We formalize functionality and security goals for message traceback, and detail two constructions that allow revealing a chain of forwarded messages (path traceback) or the entire forwarding tree (tree traceback). We implement and evaluate prototypes of our traceback schemes to highlight their practicality, and provide a discussion of deployment considerations.

ACM Reference Format:
Nirvan Tyagi, Ian Miers, and Thomas Ristenpart. 2019. Traceback for End-to-End Encrypted Messaging. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3354243

1 INTRODUCTION

Over a billion people communicate using end-to-end (E2E) encrypted messaging on platforms like WhatsApp [5], Signal [2], and Telegram [3]. Unfortunately, these platforms are being increasingly used for viral misinformation campaigns [19, 22, 23] in which parties send messages with misleading or false information, and encourage them to be forwarded by the recipient. Such campaigns have serious consequences, examples include lynchings [22] and impact on democratic elections [19].

In unencrypted contexts, such as Twitter or the Facebook news feed, platforms have started to combat misinformation campaigns with content moderation, tracing harmful messages sent through their network and intervening as deemed appropriate, e.g., by banning the "factory" accounts that are injecting such content into the network. But E2E encryption complicates moderation because the platform never observes plaintext content. Message franking [13, 14, 17] allows cryptographically verified content moderation, but current techniques only reveal the sender of a received message and do not by themselves help identify the source of a forwarded message. Platforms have so far relied solely on crude techniques like limiting the number of people to which any message can be forwarded [33].

In this paper, we initiate the study of tracing for E2E encrypted messaging. At a high level, tracing should allow users to report an abusive message along with supporting cryptographic key material to the messaging platform. Using this material, the platform can recover a cryptographically verifiable trace of the message, revealing the source and how the message was forwarded between users.

The first challenge is pinning down achievable and useful security goals for tracing. A priori it is unclear what tracing can provide in the presence of malicious users (with compromised clients). For example, it seems possible for a malicious user to "partition" the traceback by avoiding a built-in forwarding mechanism (thereby tricking the service into thinking it's a new message). We identify achievable goals for trace accountability, which, at a high level, ensure that no honest user can be erroneously implicated in sending, receiving, or forwarding a message. Thus tracing will always identify as the source either the true author of a message or a malicious user who partitioned the trace, a goal we formalize.

The second challenge is achieving accountable tracing with minimal impact on the E2E confidentiality goals of encrypted messaging. In most encrypted messaging systems, the platform learns that some message is being sent from one user to another, but never learns about plaintext content nor whether a message is a forward. We want to preserve these goals, with the exception that, when a message is reported, that message's plaintext and forwarding chain is revealed. Importantly, our security goals mandate that unreported messages retain privacy. We additionally want traceback schemes that provide third-party deniability (only the platform can cryptographically verify messages sent), a confidentiality goal explicitly sought by encrypted messaging systems [14].

We design two traceback schemes. The first, called path traceback, uses lightweight symmetric encryption techniques to add specially constructed tracing tags to ciphertexts. A tracing tag is an encrypted pointer to either the prior message, in the case of a forward, or a distinguished symbol in the case of a freshly authored message. The ability to decrypt the tracing tag is secret-shared across the platform and the recipient, so that only when a recipient reports the message can the platform decrypt the tracing tag and reveal information about forwards. By carefully ensuring that subsequent forwards' tracing tags form an encrypted linked list, the platform can, given an abuse report, trace back to the source of the message content.

Our second scheme is called tree traceback. It extends path traceback to additionally allow tracing forward from the original source to recover all the recipients of forwards of the message content. Tree traceback could be useful to platform operators when mitigating and cleaning up after abuse, since it allows, for example, identifying the victims of misinformation campaigns. Achieving

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354243

it is more complicated, however, in particular because we want to build an encrypted tree data structure incrementally, allowing tracing to connect all forwards of a message, including ones that will happen in the future. We nevertheless show how to achieve it.

Our two schemes are practical to deploy. By design, they work with arbitrary E2E encryption systems. They utilize only fast symmetric encryption and cryptographic hashing, and add a small number of bytes to each encrypted message. The schemes do require that the platform store a short trace tag (<100 bytes) for each encrypted message sent, which is nevertheless practical even for high-volume messaging systems. We implement a prototype of both of our schemes, report on initial performance evaluation, and detail how traceback can be easily integrated into existing E2E encryption protocols such as Signal [2].

In summary, our contributions are as follows:

- We introduce the notion of traceback for E2E encrypted messaging. We detail a formalization of tracing schemes and associated security goals, including various confidentiality and accountability goals.
- We design two secure tracing schemes, enabling tracing the path of a message or, additionally, the entire forwarding tree. We give formal analyses showing these schemes meet our security goals.
- We report on and open-source a prototype implementation of our constructions, and show how they can be easily integrated into existing E2E encryption systems.

2 SETTING AND GOALS

We consider an E2E encrypted message setting, in which a *platform* helps *users* send encrypted messages. A primary goal of such messaging services is confidentiality of user messages. While some industry [21] and academic [31, 32, 34] systems also target metadata privacy, i.e., obfuscating from the platform who is the sender or recipient (or both), we restrict attention to systems such as Facebook secret messenger, Whatsapp, and Signal (without the sealed sender feature) that reveal such metadata to platforms.

Some messaging clients allow *forwarding* of encrypted messages though others have not yet deployed forwarding features.[1] While forwarding is beneficial to legitimate users, it has also been subject to abuse by users spreading malicious content [19, 22]

In this work, we show how to augment encrypted messaging systems to allow users to *report* a malicious message. The platform can, given this report, trace the path a message took as it was forwarded across the network of users. This enables new moderation approaches, as we discuss more at the end of this section. We first discuss our confidentiality and accountability goals in more detail.

Confidentiality goals. We want to support traceback while minimizing impact on E2E confidentiality guarantees. Messages and forwarding behavior (whether an encrypted message is a forward, and from whom) should be confidential even from the platform, except in the case that the message was reported. For non-reported messages, neither the platform nor users should ever learn anything new due to the tracing functionality. In particular, we want:

- *Trace confidentiality for users*: Users should not learn any information about message paths beyond their local view of receiving and sending messages.
- *Pre-report trace confidentiality for platform*: Before a report, the platform should not learn any additional information about the message path beyond communication metadata (e.g., receiver, sender, timing, message length).
- *Post-report trace confidentiality for platform*: After a report, the platform should learn the message trace and nothing more.

These confidentiality goals emanate from our intention to hew closely to the existing behavior and privacy offered by deployed E2E messaging systems. First, we have chosen to ensure that tracing does not reveal to even a malicious user if they are receiving a forwarded message or a fresh one. Here, deployed systems take different approaches and we chose the approach that maximizes compatibility. Whatsapp reveals to the recipient that a message is a forward, but does not reveal from whom [33]. Signal, in contrast, does not identify forwarded messages. By providing forwarding privacy, our tracing schemes work in both contexts.

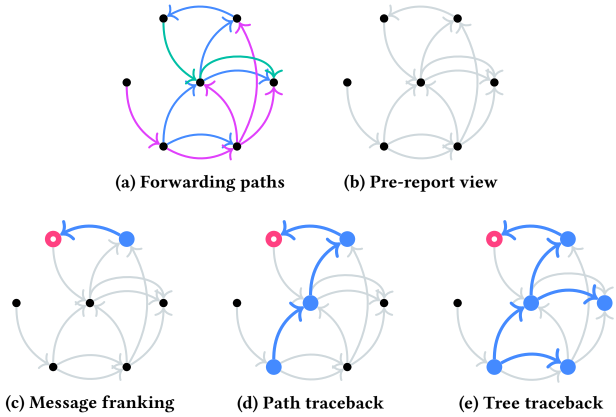
Second, we have chosen to explicitly reveal messaging meta-data. This means the platform can tell that A communicated with B who then communicated with C. Our second choice is to conceal from the platform whether B sent a fresh message to C or forwarded the message they got from A. Together these choices preserve the general property of E2E encrypted messaging systems such as WhatsApp and iMessage: the platform learns who messages who but nothing about the content of the message.

Traffic analysis by a (compromised or otherwise malicious) platform may allow inferring forwarding behavior. For example, if the platform observes a single 125 byte inbound message to some user followed shortly after by 20 outbound messages of size 125 bytes, it is likely a sequence of forwards. We will not attempt to prevent such traffic analysis, which would require expensive padding and timing obfuscations.

Finally we note that our schemes will not interfere with cryptographic deniability. In the messaging setting, this refers to the idea that recipients should not be able to provide to a third party cryptographic evidence proving that the sender sent a particular plaintext. For example, providing a recipient with a digital signature of a plaintext using the sender's private key would violate our goals. Deniability was explicitly sought by Facebook's message franking system [14], and by using symmetric primitives we will achieve whatever level of deniability is offered by the underlying E2E encryption mechanisms.

Accountability goals. In this paper, we propose two different types of traceback. The simplest form is *path traceback* which allows a platform to trace a reported message back to its origin, identifying every forward along the path. This allows the originator to be held accountable for a message that is forwarded. Separately, we consider *tree traceback*, which allows messages to be traced both back to their origin and to identify all forwards of the message. This enables not only the identification of the sender, but also of recipients. It may be useful for helping notify users about malicious

[1] Signal allows forwarding with no indication the message is a forward, WhatsApp includes an indicator, and iMessage, Telegram, and Facebook Messenger do not currently support forwarding in encrypted chats.



(a) Forwarding paths (b) Pre-report view

(c) Message franking (d) Path traceback (e) Tree traceback

Figure 1: Example of message forwards and what different traceback mechanisms report. Nodes are users and edges are message sends, with forwarding paths denoted in the same color. Subfigures (c), (d), and (e) depict what is revealed to the platform when the blue message is reported by the user indicated by the red circle.

content, or blocking further forwards of the message. A particular complication for tree traceback is that the platform should be able to trace all forwards including those that occur after the report is made. A visualization of the different traceback policies are shown in Figure 1. Shown also there is what message franking [13, 14, 17] supports in terms of accountability. Our path and tree traceback can be seen as generalizations of message franking.

As was done previously with message franking, we want to ensure various accountability properties even in the face of malicious users. But our setting is more complicated because tracing requires reasoning about multiple messages sent, as opposed to a single one. Intuitively, we require that no colluding set of adversarial users can frame an honest user as having performed some action (sending, receiving, or forwarding a message with some particular plaintext content) that they did not, in fact, perform. This implies, for example, the following accountability goals:

- *Trace unforgeability*: No group of colluding users can generate a report implicating an honest user as having performed an action they did not, in fact, perform.
- *Sender binding*: No user can author a message that cannot be traced back to them.

Note that trace unforgeability can be seen as a generalization of the receiver binding property targeted by message franking.

There are some limits to the level of accountability that we target (and that our eventual schemes will provide). In particular, we allow for malicious users to partition the traceback. In these “partition attacks”, a malicious user can split a forwarding chain so as to make themselves appear as the end of one side of the split and the source of the other side. This seems fundamentally unavoidable. Consider that a malicious user can bypass the cryptographic forwarding mechanism and emulate a forward by resending a message via a copy-paste of the received plaintext. In theory, the client software could be modified to flag such copy-paste forwards by comparing incoming and outgoing plaintexts (after decryption and before encryption, respectively). Detection is not straightforward, particularly if one wants it to be robust to small changes in the message (see Section 8 for more discussion). Moreover, this kind of defense relies upon the integrity of the client software, as an adversary using a compromised client can avoid client-side detection logic and directly submit encrypted content to the platform. In Section 8, we discuss how future work might target prevention of partition attacks despite compromised clients through the use of more heavyweight tools such as trusted hardware and expensive zero-knowledge proofs.

For the purposes of this work, we require only that a malicious user will be left holding “both ends” of a partitioned chain. That is, there will be a trace including the malicious user as having received the original message and a (possibly different) trace including the malicious user as having sent the message. In the end, this means that we will guarantee messages will be traceable either to the original source *or* to a malicious user that partitioned the chain.

The effectiveness of tracing. A final question arises: if it is possible for a user to bypass our proposed tracing protections via a hacked client, is tracing effective? We believe so. Abuse mitigation techniques need not be perfect to be effective, and even just reducing the amount of abuse is worthwhile. There is empirical evidence that protections that can be bypassed even by simple copy-paste behavior can still be effective. WhatsApp ran an experiment on the effectiveness of limiting the number of forwards a user could make of a given message. They found that this was effective despite the fact that the user could circumvent the limit after they reached it by simply copy and pasting the message. As a result, this forward limitation is now deployed globally [33].

Deployment, policy, and ethical considerations. Tracing can be used to identify the source of malicious or harassing messages. But like most technologies for content moderation, the same tools are at risk of use for silencing whistleblowers, activists, or others producing socially valuable content. Once the source account of a message is identified, they may face reprisals. We therefore believe care should be taken when deploying tracing schemes lest they themselves become abused by authoritarian regimes or others.

A material difference between tracing and prior approaches is that a reporter can implicate people with whom they did not directly communicate (i.e., the source of the message). Platforms can deploy appropriate policies and cryptographic safeguards in order to prevent abuse of the additional tracing functionality. We envision an architecture in which automated and human moderator pipelines can be used to first decide whether tracing functionality should be used on reported content. Policy could dictate use of tracing only in appropriate cases (e.g., terrorist propaganda, child pornography) versus other reported content for which less drastic actions than tracing might suffice (e.g., punitive measures against the immediate sender). Platform policy could also dictate deleting the tracing ciphertexts needed to perform traceback after some specified period of time. The challenge of developing such policies and building moderator pipelines that can enact them will need future research; we focus on showing the cryptographic viability of performing secure tracing when it is appropriate.

3 TRACING SCHEMES

In this section, we present the syntax and semantics we will use to describe message tracing schemes.

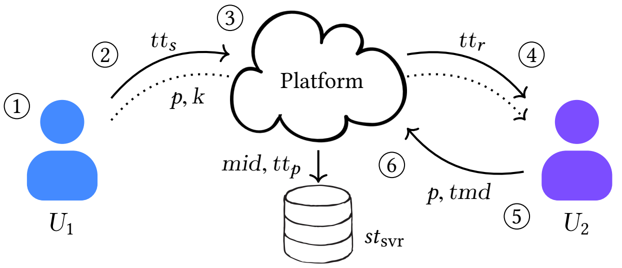
We assume users $U_1, U_2, \ldots, U_n$ each represented by a unique identifier taken from some set $\mathcal{U}$. For convenience later, we assume a distinguished user identifier $\perp$ that no real user can use. We make minimal assumptions on user identities, assuming only that they are unique and that the platform can authenticate them. In practice one will use the identifiers already used in E2E messaging systems.

We use the term *message* to refer to the sending of some plaintext from one user to another at some point in time. Multiple messages may have the same plaintext (e.g., because someone forwards message plaintext or sends the same message to multiple people). The distinction between message and plaintext will be particularly critical in our discussion of tracing scheme accountability properties (Section 5.2).

Our formalization of a message tracing scheme is decoupled from the underlying end-to-end encryption. This leads to a modular and flexible deployment path in that any message tracing scheme can be used in conjunction with any (non-metadata-private) end-to-end encryption algorithm.

A message tracing scheme MT = (NewMsg, TagGen, RecMsg, Svr-Process, Svr-Trace) is a tuple of algorithms. The first three algorithms are called by users when sending and receiving messages. The last two algorithms are for the platform to trace messages given the proper user-provided key material.

- $tmd \leftarrow\!\!\$\ \mathsf{NewMsg}(U, p)$: The randomized message authorship algorithm takes in a user, a message plaintext, and outputs trace metadata to be associated with this particular authored message instance.
- $(k, tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}(U_s, U_r, p, tmd)$: The randomized tag generation algorithm takes in the sender U_s and recipient U_r identities, a message plaintext, and trace metadata. The algorithm outputs a sender trace tag tt_s and tracing key k. The tracing key is to be included with the plaintext in the end-to-end encrypted ciphertext which is sent along with the sender trace tag to the recipient over the platform.
- $tmd \leftarrow \mathsf{RecMsg}(k, U_s, U_r, p, tt_r)$: The tag receive algorithm takes in a key, the sender U_s and recipient U_r identities, a message plaintext, and a recipient trace tag, then outputs trace metadata that cryptographically identifies the received message. The algorithm may return an error symbol $\perp$ (e.g., in the case tt_r is malformed).
- $((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Svr\text{-}Process}(st_{\mathrm{svr}}, U_s, U_r, tt_s)$: The server processing algorithm takes in the server state, the sender and receiver identities, and a sender trace tag. It outputs a recipient trace tag tt_r to deliver to the recipient as well as a message identifier mid and a platform trace tag tt_p. In our schemes, the server updates st_{svr}, which is a simple key-value store, to include (mid, tt_p).
- $tr \leftarrow \mathsf{Svr\text{-}Trace}(st_{\mathrm{svr}}, U, p, tmd)$: A user can report a received message by sending the plaintext p and trace metadata for the message to the platform. The server tracing algorithm takes in the server state, the reporting user identity, the message plaintext, and trace metadata. It then returns a trace tr of the reported message instance, the detailed structure of which depends on the tracing goal. For path traceback it corresponds to a path with nodes labeled by users and edges labeled by message identifiers. For tree traceback, a similarly-labeled tree is returned.



1. **Select message to send**:
   a. If authoring new message, $tmd \leftarrow\!\!\$\ \mathsf{NewMsg}(U_1, p)$.
   b. If forwarding message, $tmd \leftarrow \mathsf{RecMsg}(k', U_0, U_1, p, tt'_r)$.
2. **Generate trace tag**: U_s generates tracing key and sender trace tag. The tracing key and plaintext are end-to-end encrypted and sent along with the trace tag, $(k, tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}(U_1, U_2, p, tmd)$.
3. **Process trace tag**: Platform logs a message identifier and tracing data tt_p, and derives recipient trace tag tt_r for U_2, $((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Svr\text{-}Process}(st, U_1, U_2, tt_s)$.
4. **Receive trace tag**: Recipient decrypts end-to-end ciphertext and with received trace tag generates trace metadata for future forwards and reports of message, $tmd \leftarrow \mathsf{RecMsg}(k, U_1, U_2, p, tt_r)$.
5. **Report message**: Recipient sends message plaintext and trace metadata to platform, p, tmd.
6. **Trace message**: Platform learns trace of message associated with reported trace metadata, $tr \leftarrow \mathsf{Svr\text{-}Trace}(st_{\mathrm{svr}}, U_2, p, tmd)$.

Figure 2: Usage of message tracing algorithms. Solid arrows represent values passed to and handled by the platform. The dotted arrow represents the end-to-end encrypted channel between users.

Usage. The algorithms for tracing described above are designed to be decoupled from the end-to-end encryption algorithms used by the messaging platform. A typical message is sent in the following manner, depicted in Figure 2. First, the sender must specify the message they wish to send, i.e., whether it is a new message or a forward. In either case we want to associate some trace metadata to the message. If the user authors their own message, this metadata is created using NewMsg. Otherwise, RecMsg generates trace metadata for a previously received message that can be used when forwarding. To send a message, the sender generates a tracing key k and a sender trace tag tt_s using TagGen with the appropriate trace metadata. The sender encrypts the tracing key and message plaintext using the E2E encryption protocol, and sends the resulting ciphertext along with tt_s to the platform.

The platform processes tt_s using Svr-Process, updating its internal state to log a message identifier mid and associated platform trace tag tt_p. Note that tt_p does not necessarily equal tt_s. It also derives a recipient trace tag tt_r and sends the E2E ciphertext and tt_r to the recipient. The recipient decrypts the ciphertext to recover the tracing key k and plaintext, and then uses RecMsg to both verify the received trace tag and generate the trace metadata that can be used to forward the message in the future. The recipient may report a message to the platform by sending the message plaintext and associated trace metadata to the platform. The platform uses Svr-Trace with its internal state to learn a trace of the reported message instance.

Correctness. Informally, correctness dictates that trace tags created with honest calls to NewMsg, TagGen, and RecMsg and processed by an honest platform using Svr-Process should (1) not fail well-formedness verification in RecMsg, and (2) provide the correct trace with Svr-Trace when reported. Correctness is therefore context dependent, and we will discuss it more in subsequent sections. We just note that most of our schemes will not be perfectly correct, but rather be correct with all but negligible probability.

Preliminaries. Our schemes will make use of a collision-resistant pseudorandom function (CR-PRF), which we will denote as F. A CR-PRF $F : \mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ is defined over a key space $\mathcal{K}$, input space $\mathcal{X}$, and output space $\mathcal{Y}$. We will make use of two security properties of CR-PRFs. Pseudorandomness ensures the CR-PRF acts as a random function when its key remains secret, and collision resistance means it is hard to find key, input pairs that evaluate to the same output. The formalization of these security properties is deferred to the appendix, given in Figures 12 and 14. In the body, we will often drop the prefix and simply refer to F as a PRF.

4 PATH TRACEBACK

We start with path traceback. The goal is to allow reporting a message with plaintext p, with the platform then able to identify the sequence of forwarded messages back to the original author of p. In this case, Svr-Process outputs a trace

$$tr = (tr_1, mid_{1,2}, tr_2, mid_{2,3}, tr_3, \ldots, mid_{\tau-1,\tau}, tr_\tau)$$

where τ is called the trace length and each $tr_i \in \mathcal{U}$ identifies a user and each $mid_{i,j}$ is an identifier for a message. These message identifiers correspond to the ones output by the platform tag processing algorithm (Svr-Process), allowing the platform to store, and later recover during traceback, any desired metadata associated with a sent message. This can be visualized as a directed graph where nodes are associated to users and edges to messages. The trace can then be denoted via

$$p \, : \, tr_1 \xrightarrow{mid_{1,2}} tr_2 \xrightarrow{mid_{2,3}} \cdots \xrightarrow{mid_{\tau-1,\tau}} tr_\tau$$

where p represents the plaintext traced and the arrow diagram the path.

As discussed in Section 2, an adversarial user can always obfuscate the source from which they received a message by a partition attack, in which case path traceback will result in identifying the first misbehaving user (from the end) as the originator. For example, if tr_2 behaved maliciously, they can deviate from the proper client implementation and prevent traceback from identifying tr_1, and instead tr_2 would be considered the source of the message.

The linked tags scheme. Each message sent between two users is associated with a *message identifier*, denoted by mid. The message identifier is chosen by the sender, who samples a random *tracing key* k and calculates the message identifier as the output of a PRF on the plaintext, $F_k(p)$. In this manner, the message identifier also acts as a commitment to the plaintext, and the tracing key acts as an opening key. Looking forward, our trace unforgeability property will rely on the collision resistance of the PRF to bind message identifiers to a plaintext and tracing key. To link the message as a forward of a previous message, the sender also encrypts the previous message's tracing key with the tracing key for the new message. If the message is not a forward, the sender samples and encrypts a random value. This ciphertext acts as an encrypted pointer to the previous message's identifier. The current message identifier and the encrypted pointer are sent to the platform and are stored in a key-value table in server state, keyed by the message identifier. The platform sends the message identifier to the recipient, who verifies the commitment is well-formed with respect to the tracing key and plaintext before accepting the message.

Traceback is then simply a matter of decrypting and following the pointers between message identifiers in server state. Given a report consisting of a tracing key k_τ and plaintext p, the platform will lookup $mid_{\tau-1,\tau} = F_{k_\tau}(p)$ in server state and decrypt the encrypted pointer to learn the tracing key $k_{\tau-1}$. Tracing key $k_{\tau-1}$ is in turn used to lookup $mid_{\tau-2,\tau-1} = F_{k_{\tau-1}}(p)$, the previous message in the forwarding chain. The chain ends when a lookup of k_1 fails, i.e., the value $mid = F_{k_1}(p)$ is not found in the server state. Pseudocode for the construction and a diagram of one step of traceback is given in Figure 3.

Our scheme can be thought of as a sort of secret share between the platform and the recipient. The recipient gets the tracing key and the plaintext, while the platform gets the ciphertext containing the previous message's tracing key. User trace confidentiality is preserved from the recipient's share as it has no dependence on the previous message. Platform trace confidentiality is preserved from the platform's share as the message identifier and ciphertext appear as random bytes without knowledge of the tracing key. The two shares combined allow for the previous message's tracing key to be decrypted and traceback to proceed.

The linked tags scheme provides path traceback cheaply. It does require $O(m)$ storage at the platform for m the total number of messages sent by users. But storage is relatively cheap, and this is a write-heavy workload, potentially allowing cheaper storage options. Of course, the platform can expunge tracing tags after a predefined time (e.g., one week or one month), allowing tracing in the interim but not after. This may be preferable since it improves confidentiality in the long term, but still allows platforms to respond to pressing issues such as an ongoing misinformation campaign targeting candidates within an election.

5 SECURITY OF PATH TRACEBACK

5.1 Confidentiality

We start by formalizing notions of security capturing our confidentiality goals. Recall that our confidentiality goals include: (1) trace confidentiality from the platform, meaning the platform learns nothing about message contents or message history unless a report

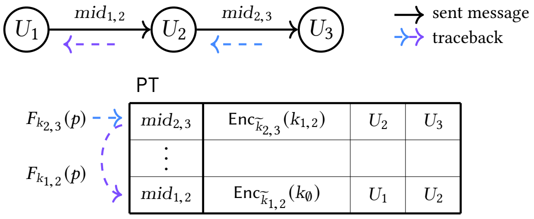
```
NewMsg($U$, $p$):
  $k \leftarrow\!\!\$ \{0,1\}^n$
  return $k$

TagGen($U_s$, $U_r$, $p$, $k_{i-1}$):
  $k_i \leftarrow\!\!\$ \{0,1\}^n$
  $\widetilde{k}_i \leftarrow H(k_i)$
  $mid \leftarrow F_{k_i}(p)$
  $ct \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(k_{i-1})$
  $tt_s \leftarrow (mid, ct)$
  return $(k_i, tt_s)$

RecMsg($k_i$, $U_s$, $U_r$, $p$, $tt_r$):
  $mid \leftarrow tt_r$
  if $mid \neq F_{k_i}(p)$ : return $\perp$
  return $k_i$

Svr-Process(PT, $U_s$, $U_r$, $tt_s$):
  $(mid, ct) \leftarrow tt_s$
  if $mid \in$ PT : return $\perp$
  $tt_p \leftarrow (ct, U_s, U_r)$
  $tt_r \leftarrow mid$
  return $((mid, tt_p), tt_r)$

Svr-Trace(PT, $U$, $p$, $k$):
  init list Tr; $i \leftarrow 0$
  Tr[$i$] $\leftarrow U$
  $mid \leftarrow F_k(p)$
  while $mid \in$ PT :
    $(ct, U_s, U_r) \leftarrow$ PT[$mid$]
    if $U_r \neq$ Tr[$i$] : break
    Tr[$i+1$] $\leftarrow mid$; Tr[$i+2$] $\leftarrow U_s$
    $\widetilde{k} \leftarrow H(k)$; $k \leftarrow \mathsf{Dec}_{\widetilde{k}}(ct)$
    $mid \leftarrow F_k(p)$
    $i \leftarrow i+2$
  return $\mathsf{Tr}^{-1}$
```



$mid_{2,3}$	$\mathsf{Enc}_{\widetilde{k}_{2,3}}(k_{1,2})$	U_2	U_3
$\vdots$			
$mid_{1,2}$	$\mathsf{Enc}_{\widetilde{k}_{1,2}}(k_\emptyset)$	U_1	U_2

Figure 3: Linked tags construction for path traceback. The diagram shows one step of traceback propagation. Solid arrows denote sent messages, while dashed arrows denote the propagation of traceback from blue to purple given the tracing key for blue, $k_{2,3}$.

implicates that message, and (2) trace confidentiality from users, meaning a user learns nothing about the history of messages they receive. We therefore formalize two notions of confidentiality.

Our confidentiality definitions isolate what might leak from the output of a specific honest node, even for adversarially chosen keys, tracing information, and messages. This ensures confidentiality goals even in more complicated attack settings, as well, for example distinguishing between a sequence of forwards and a sequence of new messages being sent.

Platform trace confidentiality. For platform trace confidentiality, we propose a real-or-random definition for the platform view, i.e., the sender trace tag of a sent message. By using a real-or-random style definition, we capture both goals of platform traceback, hiding message content and hiding message history, within a single definition. In this game, given in Figure 4 (left), the adversary $\mathcal{A}$ plays the role of the platform and is provided with a tag generation challenge oracle that either returns the trace tag output from TagGen or a random string. The task of the adversary is to distinguish between the two where an adversary's advantage is defined as

$$\mathbf{Adv}^{\text{p-tr-conf}}_{\text{MT}}(\mathcal{A}) = \Big|\Pr\Big[\text{PTrCONF}^{\mathcal{A},1}_{\text{MT}} \Rightarrow 1\Big] - \Pr\Big[\text{PTrCONF}^{\mathcal{A},0}_{\text{MT}} \Rightarrow 1\Big]\Big| .$$

The sender trace tag in our path traceback scheme is made up of a message identifier, which is the output of a PRF, and a ciphertext. Intuitively, since the platform does not learn the key used with the PRF or with encryption scheme, our scheme satisfies the security property. More formally,

Theorem 1. *Let* MT *be the message tracing scheme for path traceback defined in Figure 3 using hash function H. Then if H is modeled as a random oracle, for any* PTrCONF *adversary $\mathcal{A}$ that makes at most q oracle queries, we give adversary $\mathcal{B}$ and C such that*

$$\mathbf{Adv}^{\textit{p-tr-conf}}_{\text{MT}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F,q}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror-cpa}}_{\mathcal{E},q}(C)$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ and C run in time $T' \approx T$ and $\mathcal{B}$ makes at most q oracle queries.

In the above, we use ≈ to hide small constants. We defer the full proof as well as full definitions for the reduction targets to Appendix A, and provide a sketch here.

Proof sketch: The proof proceeds in a straightforward fashion through two main game hops. The first replaces the PRF evaluation with that of a random function and bounds the distinguishing advantage by the PRF security of F. The second replaces the encryption output with a random bit string and bounds the distinguishing advantage by the real-or-random security of the underlying encryption scheme. After these two steps, it is easy to see that the sender trace tag, which consists of the output mid of F and a ciphertext ct, is a random bit string.

User trace confidentiality. In user trace confidentiality, a real-or-random style definition will not work, as the recipient's view of tracing key, plaintext, and recipient trace tag have a related structure and must verify under RecMsg. Instead, we focus on the specific goal we aim to achieve under user trace confidentiality, namely that message history is not revealed. We thus task the adversary with distinguishing between the result of an authored message and a forwarded message. The adversary gets to choose the plaintext and, in the case the challenge oracle is forwarding, the tracing key and recipient trace tag representing the message to be forwarded. The game pseudocode is given in Figure 4 (right). We define the distinguishing advantage of the adversary as:

$$\mathbf{Adv}^{\text{u-tr-conf}}_{\text{MT}}(\mathcal{A}) = \Big|\Pr\Big[\text{UTrCONF}^{\mathcal{A},1}_{\text{MT}} \Rightarrow 1\Big] - \Pr\Big[\text{UTrCONF}^{\mathcal{A},0}_{\text{MT}} \Rightarrow 1\Big]\Big| .$$

In our path traceback scheme, the recipient's view consists of the message identifier mid, tracing key k_i, and the plaintext. Importantly, the ciphertext ct is stored by the platform and not visible to the recipient. The tracing key is randomly generated for each sent message and the message identifier is calculated as a function of the tracing key and plaintext, $mid \leftarrow F_{k_i}(p)$. Thus, no part of the recipient's view is dependent on previous message trace metadata, e.g., k_{i-1}, and any adversary's advantage against our scheme is 0.

Theorem 2. *Let* MT *be the message tracing scheme for path traceback defined in Figure 3. For any* UTrCONF *adversary $\mathcal{A}$,*

$$\mathbf{Adv}^{\textit{u-tr-conf}}_{\text{MT}}(\mathcal{A}) = 0 .$$

```
$\mathrm{PTrCONF}^{\mathcal{A},b}_{\mathrm{MT}}$:
$b' \hookleftarrow\$ \mathcal{A}^{\mathrm{Chal}}$
return $b'$

$\mathrm{Chal}(U_1, U_2, tmd, p)$:
$(k, tt^1_s) \hookleftarrow\$ \mathsf{TagGen}(U_1, U_2, p, tmd)$
$tt^0_s \hookleftarrow\$ \{0,1\}^{\mathrm{len}(tt_s)}$
return $tt^b_s$
```

```
$\mathrm{UTrCONF}^{\mathcal{A},b}_{\mathrm{MT}}$:
$b' \hookleftarrow\$ \mathcal{A}^{\mathrm{Chal}}$
return $b'$

$\mathrm{Chal}(st, U_0, k, tt_r, U_1, U_2, p)$:
$tmd_0 \leftarrow \mathsf{RecMsg}(k, U_0, U_1, p, tt_r)$
if $tmd_0 = \bot$ then return $\bot$
$tmd_1 \hookleftarrow\$ \mathsf{NewMsg}(U_1, p)$
$(k', tt_s) \hookleftarrow\$ \mathsf{TagGen}(U_1, U_2, p, tmd_b)$
$((mid, tt_p), tt'_r) \leftarrow \mathsf{Svr\text{-}Process}(st, U_1, U_2, tt_s)$
return $(k', tt'_r)$
```

Figure 4: Notions for (left) platform trace confidentiality and (right) user trace confidentiality.

5.2 Accountability

Tracing should accurately identify the source of a message, but malicious users can always obfuscate from whom they've received a message. We therefore want tracing never to result in an honest user erroneously implicated in having taken an action (sent, forwarded, or received a message) they did not, in fact, perform.

To formalize accountability we use a game-based approach in which an adversary interacts with some number n of honest users. See Figure 5. The adversary can cause honest users to author and send (adversarially chosen) messages via two oracles **NewMsg** and **Send**. The adversary can also pose as any number of malicious users, sending messages via a malicious send oracle **SendMal**. We identify users by a number, user i is honest if $i \in [1, n]$ and user i is malicious if $i \notin [1, n]$. In our exposition we often use variables $U_1, \ldots, U_n$ to refer to the honest users, and A_j for $j \notin [1, n]$ for a malicious user. The security experiment here assumes authenticated channels — the adversary cannot send messages as an honest user nor manipulate messages sent between the honest users and the platform. On the other hand, we give the adversary the power to observe trace tags generated by, or sent to, honest users, but they only see the tracing keys sent by honest parties to malicious users. Given the use of secure channels to send messages, it would seem sufficient to not reveal communications from honest parties to the platform and from the platform to honest parties, but giving the adversary this information only makes the security achieved stronger.

The adversary's goal is to generate a report that results in an *invalid* trace, one that indicates that an honest user took some action that they did not, in fact, take. Note that the adversary can either have an honest user or malicious user make a report. In the honest case, the adversary outputs a value c^* indicating which message received by i^* is being reported, and in the malicious case the adversary directly outputs an opening k^*. The game loops over the reported trace (skipping the loop entirely if Svr-Trace output an error), and checks for each honest user implicated in the trace whether the reported trace matches an action they in fact performed. We do this via a set of predicates, corresponding to where in the trace the honest user appears, and whether they actually received and/or sent the indicated messages.

As a non-exhaustive list of example invalid traces ruled out by these predicates, consider the following scenarios, where for simplicity we use a single honest user ($n = 1$):

```
$\mathrm{TrUNF}^{\mathcal{A}}_{\mathrm{MT},n}$:
$c \leftarrow 0$
$(i^*, p^*, k^*, c^*) \hookleftarrow\$ \mathcal{A}^{O}$
if $i^* \in [1, n]$ then $p^* \leftarrow p_{c^*}$; $k^* \leftarrow k_{i^*,c^*}$
$(tr_1, mid_{1,2}, tr_2, \ldots, mid_{\tau-1,\tau}, tr_\tau) \leftarrow \mathsf{Svr\text{-}Trace}(\mathsf{PT}, i^*, p^*, k^*)$
if $(i^* \in [1, n] \wedge \tau = 1)$ then return true
for $j \in 1$ to $\tau$ do
  if $tr_j \in [1, n]$ then
    if    $\big(j = 1 \wedge (\mathsf{WasSent}(tr_1, tr_2, mid_{1,2}, p^*) = \mathsf{false}\big)$
       or $\big(j = \tau \wedge \mathsf{WasRec}(tr_{\tau-1}, tr_\tau, mid_{\tau-1,\tau}, p^*) = \mathsf{false}\big)$
       or $\big(1 < j < \tau \wedge \mathsf{WasFwd}(tr_{i-1}, tr_i, mid_{i-1,i}, mid_{i,i+1}, p^*) = \mathsf{false}\big)$ then
      return true
return false

$\mathbf{NewMsg}(i, p)$:
if $i \notin [1, n]$ then return $\bot$
$c \leftarrow c + 1$
$p_c \leftarrow p$
$k_{i,c} \hookleftarrow\$ \mathsf{NewMsg}(i, p_c)$
$mid_c \leftarrow \mathsf{auth}$
return

$\mathbf{Send}(i, j, s)$:
if $i \notin [1, n] \vee k_{i,s} = \bot$ then return $\bot$
$(k, tt_s) \hookleftarrow\$ \mathsf{TagGen}(i, j, p_s, k_{i,s})$
$((mid, tt_p), tt_r) \hookleftarrow\$ \mathsf{Svr\text{-}Process}(\mathsf{PT}, i, j, tt_s)$
$\mathsf{PT}[mid] \leftarrow tt_p$
if $mid_s = \mathsf{auth}$ then $\mathsf{WasSent}(i, j, mid, p_s) \leftarrow \mathsf{true}$
else $\mathsf{WasFwd}(i, j, mid_s, mid, p_s) \leftarrow \mathsf{true}$
if $j \in [1, n]$ then
  $c \leftarrow c + 1$
  $p_c \leftarrow p_s$
  $k_{j,c} \leftarrow \mathsf{RecMsg}(k, i, j, p_c, tt_r)$
  $\mathsf{WasRec}(i, j, mid, p_s) \leftarrow \mathsf{true}$
  $mid_c \leftarrow mid$
  return $tt_r$
return $(tt_r, k)$

$\mathbf{SendMal}(i, j, k, p, tt_s)$:
if $i \in [1, n]$ then return $\bot$
$((mid, tt_p), tt_r) \hookleftarrow\$ \mathsf{Svr\text{-}Process}(\mathsf{PT}, i, j, tt_s)$
$\mathsf{PT}[mid] \leftarrow tt_p$
if $j \in [1, n]$ then
  $k \leftarrow \mathsf{RecMsg}(k, p, tt_r)$
  if $k \neq \bot$ then
    $c \leftarrow c + 1$
    $p_c \leftarrow p$
    $k_{j,c} \leftarrow k$
    $\mathsf{WasRec}(i, j, mid, p) \leftarrow \mathsf{true}$
    $mid_c \leftarrow mid$
return $tt_r$
```

Figure 5: Trace unforgeability security game for path traceback.

- ***Message replacement***: Honest user U_1 sends a message mid with plaintext p to a malicious user A_2, who then successfully reports the trace $p^* : U_1 \xrightarrow{mid} A_2$ for some plaintext $p^* \neq p$. This frames the honest user as having sent the wrong plaintext. The only valid trace in this case is $p : U_1 \xrightarrow{mid} A_2$.
- ***Identity replacement***: Honest user U_1 sends a message mid with plaintext p to a malicious user A_2, who then successfully reports the trace $p : U_1 \xrightarrow{mid} A_3$ for some distinct user A_3. This frames the honest user as having sent the message to a different user. The only valid trace in this case is $p : U_1 \xrightarrow{mid} A_2$.

- ***Path suffix***: Malicious user A_2 sends a message mid_a with plaintext p to the honest user U_1, and then U_1 forwards mid_a to another user A_3 in message mid_b. Then A_3 successfully reports the trace $p : U_1 \xrightarrow{mid_b} A_3$. This frames the honest user as having originated a message that they instead forwarded from someone else. The valid traces that can be reported in this case are $p : A_2 \xrightarrow{mid_a} U_1$ and $p : A_2 \xrightarrow{mid_a} U_1 \xrightarrow{mid_b} A_2$.
- ***Same-message, wrong path***: Two malicious users A_2, A_3 send messages mid_a, mid_b with the same plaintext p to the honest user U_1. The honest user forwards the message mid_a from A_2 to a user A_4 in a message mid_c. Finally A_4 generates a report resulting in trace $p : A_3 \xrightarrow{mid_b} U_1 \xrightarrow{mid_c} A_4$. This frames the honest user as having forwarded a different message, despite the plaintext being the same this could be an accountability problem given that the sender and message time are incorrect. The valid traces that can be reported in this case are $p : A_2 \xrightarrow{mid_a} U_1$, $p : A_3 \xrightarrow{mid_b} U_1$, and $p : A_2 \xrightarrow{mid_a} U_1 \xrightarrow{mid_c} A_4$.

Notice that the prefix of any valid trace is also a valid trace (though the reporter would be different in each case), but suffixes of a valid trace are *not* always valid (second example). Also the examples highlight the importance of tracing particular messages, not just plaintexts, as we want the platform to be able to reliably associate metadata (senders, receivers, timing) of messages to a reported trace.

We associate to any tracing scheme MT, number of honest users n, and adversary $\mathcal{A}$ the path traceback forging advantage

$$\mathbf{Adv}^{\text{tr-unf}}_{\text{MT},n}(\mathcal{A}) = \Pr\left[\text{TrUNF}^{\mathcal{A}}_{\text{MT},n} \Rightarrow \text{true}\right]$$

where the probability is taken over the random choices made in the game, including those made by the adversary.

Intuitively, trace unforgeability is achieved in our path traceback scheme due to the binding of the plaintext to a message identifier with the tracing key. Honest users check the binding of message identifiers they receive and send, so an adversary that wishes to frame a user must find a collision on one of the honest user's message identifiers. For example, to achieve the message replacement attack described above, an adversary must find an alternate plaintext and key that collides with the honest user's sent message identifier. We thus provide the following theorem statement:

Theorem 3. *Let* MT *be the message tracing scheme for path traceback defined in Figure 3. Then, for any TrUNF adversary $\mathcal{A}$ that makes at most q_{nm} new message queries, q_{s} send queries, and q_{sm} malicious send queries, we give adversaries $\mathcal{B}$ and C such that*

$$\mathbf{Adv}^{\textit{tr-unf}}_{\text{MT},n}(\mathcal{A}) \le \mathbf{Adv}^{\textit{prf}}_{F,q_{\text{nm}}}(\mathcal{B}) + \mathbf{Adv}^{\textit{cr}}_{F}(C) + \frac{q_{\text{s}} + q_{\text{sm}}}{2^n}$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ and C run in time $T' \approx T$ and $\mathcal{B}$ makes at most $q_{\text{s}} + q_{\text{sm}}$ oracle queries.

We build a reduction using the collision resistance and PRF security of F. We defer the full proof as well as full definitions for the reduction targets to Appendix B, and provide a sketch here.

Proof sketch: This proof proceeds as a careful case analysis of the four adversary winning conditions. We show that nearly all of the winning conditions correspond to an adversary finding a collision in F. In the single subcase that does not result in a collision, we argue that the probability of reaching this subcase corresponds to guessing the output of F keyed by a tracing key sampled in **NewMsg**. Since tracing keys sampled in **NewMsg** are never revealed to the adversary, we can show this probability is low using the PRF security of F.

6 TREE TRACEBACK

Next, we consider an alternative traceback goal, tree traceback. The goal in tree traceback is to allow reporting of a message with plaintext p, enabling the platform to identify not just the path of forwarded messages to the original author, as in path traceback, but the entire forwarding tree of messages for p rooted at the original author. A tree is denoted as a tuple of user identifier and list of children subtrees, where each element of the children subtree list is recursively a tree, i.e., a user identifier and list of children subtrees, along with the message identifier for the message sent between parent and child:

$$tr = \left(tr_a, \left[(mid_{a,0}, tr_b, [\ldots]), (mid_{a,1}, tr_c, [\ldots]), \ldots\right]\right)$$

where $tr_\alpha \in \mathcal{U}$ identifies a user and each $mid_{\alpha,i}$ is an identifier for a message sent by tr_α.

The doubly-linked tags scheme. The doubly-linked tags construction for tree traceback extends the strategy taken in path traceback of storing encrypted pointers between message identifiers. In the path traceback construction, for each message between a sender and recipient, the platform stored an encrypted pointer to trace backwards to the previous message, i.e., where the sender received the message from. Intuitively, in tree traceback, we need to extend this approach to also trace forwards in order to build the forwarding tree. This includes storing pointers to forwards made by the sender *and* forwards made by the recipient. However, attempting to explicitly store encrypted pointers to other forwards is problematic as the number of forwards of a message are not known at the time of sending; the recipient has not yet even received the message, let alone forwarded it, and the sender could choose to forward the message to new recipients in the future.

We address this challenge by efficiently representing an unbounded set of pointers with a PRF key, gk, which acts as a generator for all the tracing keys associated to forwards of a particular message. One enumerates the tracing keys by evaluating the PRF on a counter ctr which is stored in the client's state, $k \leftarrow F_{gk}(ctr)$. As in the path traceback construction, a tracing key points to a message identifier through the evaluation of a PRF, $mid \leftarrow F_k(p)$. Thus, the platform stores three encrypted values with each message identifier: (1) an encrypted tracing key $ct_{k_{i-1}}$ for the previous message; (2) an encrypted tracing key generator ct_{gk_i} for other forwards by the sender; and (3) an encrypted tracing key generator $(ct_{gk_{i+1}}, ks_1)$ for forwards by the recipient.

The three encrypted values stored by the platform correspond to three stages of performing tree traceback, each illustrated in Figure 6. Given a report consisting of a tracing key and plaintext, first the platform follows the tracing keys, k_{i-1}, for previous message identifiers until reaching the message source, essentially performing path traceback (Figure 6 (a)). Next, using the sender tracing key generator, gk_i, the platform enumerates the tracing keys and message identifiers for all sends of the message from the source sender (Figure 6(b)). Lastly, for each of these message identifiers, the platform recursively builds out a subtree by enumerating the tracing keys and message identifiers for the recipient's forwards using the recipient tracing key generator, gk_{i+1} (Figure 6 (c)). The full pseudocode for the scheme is also given in Figure 6.

There are a few subtle design points to our tree traceback scheme that we highlight here. The first concern is how to securely escrow the three encrypted values for each message identifier with the platform. The sender can encrypt and send the previous message's tracing key and its own tracing key generator, but the sender cannot know and therefore cannot escrow the recipient's tracing key generator. For confidentiality, the recipient's tracing key generator should only be known by the recipient. Instead, the sender and the platform each create key shares for the recipient tracing key, ks_0 and ks_1, such that neither the sender nor the platform learn the key, but the sender's key share needed to derive the key is stored encrypted on the platform. The recipient derives their tracing key generator as $gk_{i+1} \leftarrow H(ks_0 \parallel ks_1)$ which appears random (given that the sender and platform are not colluding).

A second concern arises from the enumeration of tracing keys from a generator during traceback. In stages (b) and (c) in Figure 6, the platform enumerates tracing keys by evaluating a PRF keyed by the escrowed generator, gk, on a counter initialized to zero, incrementing and re-evaluating to produce the next tracing key. This continues until the produced tracing key does not evaluate to a valid message identifier in platform storage, indicating all forwarding branches for the user have been enumerated. This traceback approach will only succeed if users correctly derive tracing key generators from the escrowed key shares and correctly derive tracing keys from the generator by incrementing a counter, and not, for example, skipping a counter value. Left as is, these types of deviations would result in a class of partition attacks that are so-called "unidirectional". In these attacks, a malicious user is able to partition the tree trace to hide a subtree such that a report in the main tree will end at the malicious user and not include the subtree; but at the same time, a report in the subtree will trace through the malicious user and identify the main tree as the source.

We address this by enforcing that a message is only traced back to a sender if it would have also been traced forward to the recipient. This invariant restricts malicious users to only being able to mount "complete" partition attacks, in which if they choose to partition, they are implicated in two disjoint traces: the end of one trace and the source of the other. Enforcing this invariant manifests in tree traceback by two well-formedness checks (denoted $\mathsf{WellFormed}_{gk}$ and $\mathsf{WellFormed}_{k}$ in the pseudocode). The first check simply rederives the recipient generator to make sure it matches the one escrowed by the recipient. The second well-formedness check determines if a tracing key was properly derived from a generator. Doing so requires enumerating with a counter, succeeding when the current message tracing key is found, or fails when a generated tracing key evaluates to an invalid message identifier. Both of these checks take place during stage (a) of tree traceback to identify a root that will not include fragmented subtrees.

Security of tree traceback. We formalize confidentiality and accountability security in Appendix C, and just briefly describe it here.

```
NewMsg($U$, $m$):
$k_{i-1} \leftarrow\!\!\$ \{0,1\}^n$
$gk_i \leftarrow\!\!\$ \{0,1\}^n$
$tmd \leftarrow (k_{i-1}, gk_i)$
return $tmd$

TagGen(CTR, $U_s$, $U_r$, $p$, $tmd$):
if $tmd \notin \mathsf{CTR} : \mathsf{CTR}[tmd] \leftarrow 0$
$ctr \leftarrow \mathsf{CTR}[tmd]$
$\mathsf{CTR}[tmd] \leftarrow ctr + 1$
$(k_{i-1}, gk_i) \leftarrow tmd$
$k_i \leftarrow F_{gk_i}(ctr)$
$mid \leftarrow F_{k_i}(p)$
$ks_0 \leftarrow\!\!\$ \{0,1\}^n$
$\widetilde{k}_i \leftarrow H(k_i)$
$ct_{k_{i-1}} \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(k_{i-1})$
$ct_{gk_i} \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(gk_i)$
$ct_{gk_{i+1}} \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(ks_0)$
$tt_s \leftarrow (mid, ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}})$
return $(\mathsf{CTR}, k_i, tt_s)$

RecMsg($k_i$, $U_s$, $U_r$, $p$, $tt_r$):
$(mid, ct_{gk_{i+1}}, ks_1) \leftarrow tt_r$
if $mid \neq F_{k_i}(p)$ : return $\perp$
$\widetilde{k}_i \leftarrow H(k_i)$
$ks_0 \leftarrow \mathsf{Dec}_{\widetilde{k}_i}(ct_{gk_{i+1}})$
$gk_{i+1} \leftarrow H(ks_0 \parallel ks_1)$
$tmd \leftarrow (k_i, gk_{i+1})$
return $tmd$

Svr-Process(PT, $U_s$, $U_r$, $tt_s$):
$(mid, ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}) \leftarrow tt_s$
if $mid \in \mathsf{PT}$ : return $\perp$
$ks_1 \leftarrow\!\!\$ \{0,1\}^n$
$tt_p \leftarrow (ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, U_s, U_r)$
$tt_r \leftarrow (mid, ct_{gk_{i+1}}, ks_1)$
return $((mid, tt_p), tt_r)$

Svr-Trace(PT, $U$, $p$, $tmd$):
$(k_i, gk_{i+1}) \leftarrow tmd$
$root \leftarrow U$
$root_{gk} \leftarrow gk_{i+1}$
$mid \leftarrow F_{k_i}(p)$
while $mid \in \mathsf{PT}$ :
  $(ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, U_s, U_r) \leftarrow \mathsf{PT}[mid]$
  if $root \neq U_r$ : break
  $\widetilde{k}_i \leftarrow H(k_i)$
  $ks_0 \leftarrow \mathsf{Dec}_{\widetilde{k}_i}(ct_{gk_{i+1}})$
  if not $\mathsf{WellFormed}_{gk}(gk_{i+1}, ks_0, ks_1)$ : break
  $gk_i \leftarrow \mathsf{Dec}_{\widetilde{k}_i}(ct_{gk_i})$
  if not $\mathsf{WellFormed}_{k}(k_i, gk_i, \mathsf{PT})$ :
    return $(U_s, [(F_{k_i}(p), \mathsf{trace\_fwd}(\mathsf{PT}, p, U_r, gk_{i+1}))])$
  $root \leftarrow U_s$
  $root_{gk} \leftarrow gk_i$
  $k_i \leftarrow \mathsf{Dec}_{\widetilde{k}_i}(ct_{k_i})$; $gk_{i+1} \leftarrow gk_i$
  $mid \leftarrow F_{k_i}(p)$
return $\mathsf{trace\_fwd}(\mathsf{PT}, p, root, root_{gk})$

trace_fwd(PT, $p$, $U$, $gk_i$):
init list Tr; $j \leftarrow 0$
while $k \leftarrow F_{gk_i}(j)$; $mid \leftarrow F_k(p)$; $mid \in \mathsf{PT}$ :
  $(ct_{k_{i-1}}, ct_{gk_i}, ct_{gk_{i+1}}, ks_1, U_s, U_r) \leftarrow \mathsf{PT}[mid]$
  $\widetilde{k} \leftarrow H(k)$
  $ks_0 \leftarrow \mathsf{Dec}_{\widetilde{k}}(ct_{gk_{i+1}})$
  $gk_{i+1} \leftarrow H(ks_0 \parallel ks_1)$
  $\mathsf{Tr}[j] \leftarrow (mid, \mathsf{trace\_fwd}(\mathsf{PT}, p, U_r, gk_{i+1}))$
  $j \leftarrow j + 1$
return $(U, \mathsf{Tr})$
```

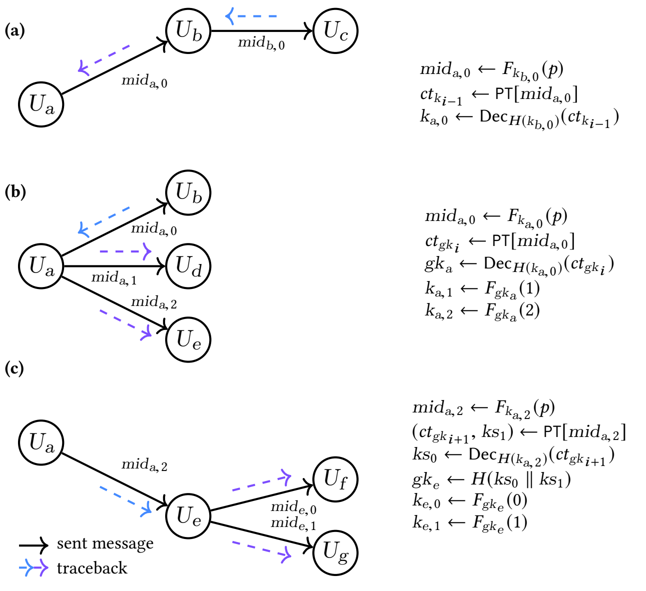

(a)
U_a
U_b
U_c
$mid_{a,0}$
$mid_{b,0}$
$mid_{a,0} \leftarrow F_{k_{b,0}}(p)$
$ct_{k_{i-1}} \leftarrow \mathsf{PT}[mid_{a,0}]$
$k_{a,0} \leftarrow \mathsf{Dec}_{H(k_{b,0})}(ct_{k_{i-1}})$
(b)
U_a
U_b
U_d
U_e
$mid_{a,0}$
$mid_{a,1}$
$mid_{a,2}$
$mid_{a,0} \leftarrow F_{k_{a,0}}(p)$
$ct_{gk_i} \leftarrow \mathsf{PT}[mid_{a,0}]$
$gk_a \leftarrow \mathsf{Dec}_{H(k_{a,0})}(ct_{gk_i})$
$k_{a,1} \leftarrow F_{gk_a}(1)$
$k_{a,2} \leftarrow F_{gk_a}(2)$
(c)
U_a
U_e
U_f
U_g
$mid_{a,2}$
$mid_{e,0}$
$mid_{e,1}$
sent message
traceback
$mid_{a,2} \leftarrow F_{k_{a,2}}(p)$
$(ct_{gk_{i+1}}, ks_1) \leftarrow \mathsf{PT}[mid_{a,2}]$
$ks_0 \leftarrow \mathsf{Dec}_{H(k_{a,2})}(ct_{gk_{i+1}})$
$gk_e \leftarrow H(ks_0 \parallel ks_1)$
$k_{e,0} \leftarrow F_{gk_e}(0)$
$k_{e,1} \leftarrow F_{gk_e}(1)$

Figure 6: Doubly-linked tags construction for tree traceback. The subdiagrams show how traceback propagates in each of three stages of tree traceback: (a) trace to parent up to root, (b) trace to siblings, (c) trace to children recursively. Solid arrows denote sent messages, while dashed arrows denote the propagation of traceback from blue to purple given tracing key for blue.

The starting point is the definitions for path traceback detailed in Section 5, but adapted to fit the tree traceback syntax and semantics. Because our clients are now stateful (due to the counters used along with key generators), we modify the confidentiality games to allow the adversary to drive the honest client to an arbitrary state before submitting challenge queries.

The accountability game is modified to iterate through the returned tree trace checking the appropriate conditions. This definition currently only guarantees that no honest user can be "framed" as having performed some action they did not, in fact, perform. This does not capture some accountability attacks that our tree traceback scheme protects against, in particular, the difference between allowing unidirectional partitions versus the stronger guarantee of only allowing complete partitions (as described above). An open question is therefore how to provide stronger formal definitions.

We provide a formal analysis of the doubly-linked tags scheme in Appendix C.

7 IMPLEMENTATION AND EVALUATION

To evaluate our tracing protocols, we provide a prototype library and tracing service implementation in Rust that can be readily integrated into existing end-to-end encrypted messaging systems. For our hash, collision-resistant pseudorandom function, and block cipher primitives, we use SHA-3, HMAC derived from SHA-3, and AES-128. All of these operations are supported by the Rust Crypto library [4]. The tracing service uses Redis [1] as its underlying key-value store. Both the library and service code are available open source at https://github.com/nirvantyagi/tracing.

Our tracing service can be integrated into existing end-to-end encrypted messaging systems with the following client and server side changes. The client will make library calls to NewMsg, TagGen, and RecMsg when sending, forwarding, and receiving messages and it will store trace metadata associated with messages accepted by RecMsg. The tracing key is included with the plaintext in the end-to-end encrypted ciphertext. The sender and recipient trace tags are sent alongside the end-to-end ciphertext to and from the platform. On the server side, the tracing service is run as an internal service. The messaging server receives the end-to-end ciphertext and sender trace tag and sends a "process" request including the sender trace tag and user identifiers for the sender and recipient to the tracing service. The tracing service handles running Svr-Process, storing the appropriate information in a key-value store, and returns the recipient trace tag, which the messaging server delivers with the end-to-end ciphertext. The client and messaging server also need to be modified to send and accept reports of messages (if that functionality is not already included). The client includes the trace metadata along with the plaintext in the report. The messaging server simply forwards the reported plaintext and trace metadata to the tracing service which runs Svr-Trace and saves the message trace to be used downstream for moderation.

Scheme	TagGen (μs)	RecMsg (μs)
path traceback	7.4 ± 0.4	1.7 ± 0.1
tree traceback	26.0 ± 2.8	7.3 ± 0.1

Figure 7: Time to run client-side tag generation and verification algorithms.

Scheme	Sender bandwidth (B)	Recipient bandwidth (B)	Platform storage (B)	Client storage (B)
path traceback	64	48	36	16
tree traceback	96	80	84	34

Figure 8: Bandwidth and storage costs incurred per message.

Timing benchmarks. At a high level, our tracing schemes are fast and induce minimal storage and bandwidth overhead. This is to be expected as our schemes are composed of symmetric cryptographic techniques over small 128 bit components. Experiments were performed on a 2.2 GHz Intel Core i7 Processor with 8 GB of RAM. The time to run the client-side algorithms, TagGen and RecMsg, for generating and verifying trace tags is shown in Figure 9; the NewMsg algorithm is not shown as it simply samples a random number. For path traceback the tag generation and verification algorithms take < 10 microseconds, and for tree traceback the algorithms take < 50 microseconds. In practice, client side operations will often be performed on less powerful mobile devices, e.g. running ARM processors, but we do not expect the difference in timing to be prohibitive.

We next turn to evaluate the server side algorithms, Svr-Process and Svr-Trace. The server side algorithms interact with a key-value store, the performance characteristics of which will affect the performance of the algorithm. In our schemes, the Svr-Process algorithm essentially performs a key-value put operation and relays the trace tag; no cryptographic operations are performed. As expected, this translates to a minimal cost operation for most key-value stores – in our benchmarks, in which the server key-value store is instantiated with an in-memory Redis data store, the Svr-Process algorithm takes on the order of 100 microseconds. Building a trace of messages using the Svr-Trace algorithm is where the majority of computation is performed. Importantly, the time to build a trace is linearly dependent on the number of messages revealed in the trace, i.e. trace size. Intuitively, this is because our schemes traverse the trace message by message performing only a constant number of decryptions and PRF evaluations per message. This relationship is easily seen for the case of path traceback as shown in Figure 9, in which we find that building a trace takes ≈ 100 microseconds per message in the path. For tree traceback, the traceback time is dependent not only on tree size, but also on tree structure; in particular, the branching factor, i.e., the average number of forwards made by each user. In our tree traceback scheme, the branching factor is the number of forward tracing keys learned per message lookup and decryption. The cost of PRF evaluations to enumerate forward tracing keys is less expensive than the key-value lookup and decryption cost of dereferencing a tracing key. Thus, as the branching factor increases, the time per message decreases (Figure 10 (right)). In the worst case, with branching factor equal to one, i.e. a path, the time per message is < 300 microseconds, which still leads to efficient tree traces regardless of structure. As an example, a trace of a tree of size 20,000 is built in under two seconds.

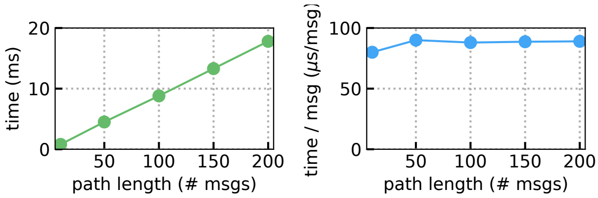


Figure 9: Path traceback timing with respect to path length. (Left) Total time to complete trace. (Right) Traceback rate of time per message in trace.

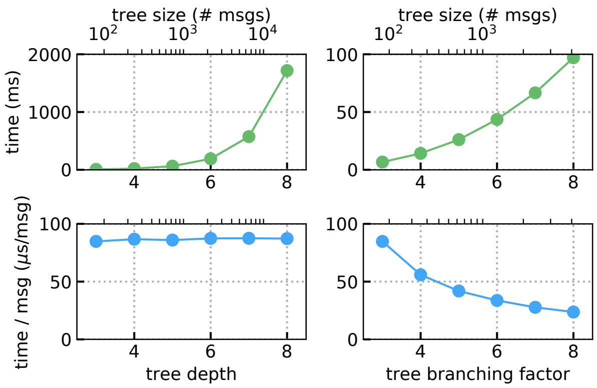


Figure 10: Tree traceback with varying tree structure. (Left) Varying tree depth with constant branching factor of 3. (Right) Varying branching factor with constant depth of 3. (Top) Total time to complete trace. (Bottom) Traceback rate of time per message in trace.

Storage and bandwidth overhead. Our tracing schemes introduce extra tracing metadata that needs to be stored and send by both the client and server. As shown in Figure 8, the absolute size of the stored trace metadata is small — a 256-bit PRF output and a few 128-bit block cipher outputs. For client storage and bandwidth, we expect the overhead induced by $<$ 100B of trace metadata per message to be dwarfed by the size of the message itself; furthermore, for client storage, when the message is deleted, the associated trace metadata can be deleted with it. For server storage, however, in platforms like Signal and WhatsApp, message ciphertexts are not stored, aside from a temporary staging period until they have been delivered. In this case, trace metadata incurs the addition of a new long-term storage cost that potentially represents a significant infrastructure change. To limit storage costs, if the goal of message tracing is to combat ongoing misinformation campaigns, it seems reasonable to store only a sliding window of trace metadata, say for the current month. In this case, if the platform sees one billion messages of traffic per day, the data store would be of size $\approx$ 2TB for tree traceback and 600GB for path traceback. A data store of this size can be instantiated with an in-memory data store like Redis as in our benchmarks, or more cheaply with a database, where the tradeoff would be slower traceback.

8 DISCUSSION

Here we present further discussion on open questions raised by this work.

Supporting more general plaintext linking policies. Recall that we hypothesized it is possible to use client-side software in honest clients to detect and prevent copy-paste behavior and unintentional partitioning, therefore limiting partition attacks to malicious users with hacked clients. One of the challenges of building such detection software is making it robust to small perturbations in messages as they are forwarded. Our schemes focus on tracing messages that contain identical plaintexts for simplicity of presentation, but our formalization and schemes can easily be extended to support more general tracing policies, such as tracing closely related message content (e.g., similar images). Given a received message with plaintext p and a message with plaintext p' to be sent, an honest client can run some arbitrary procedure to determine whether to allow (or require) tracing to link p and p'. The procedure could also take into account the sender and recipient identities. Supporting traceback in this context will require escrowing with the platform an encryption of p (or, at least, a diff between p and p'), which will inflate storage costs. This could allow significantly more flexibility in tracing functionality, with the accompanying risk of more permissive revelation of plaintext content during tracing. Deployment of such a policy would require defining actionable policies about what messages should be considered related; a tricky topic that requires more research.

Preventing partition attacks by malicious users. If we assume the existence of effective message-similarity detection software running on honest clients, one solution to prevent malicious users from bypassing these checks is trusted hardware; i.e., require all clients to faithfully execute the detection software. In the absence of client-side trusted hardware, we can attempt to prevent such behavior cryptographically using zero-knowledge proofs, albeit at a far greater cost. Such a solution would require the platform to maintain, for each user, a set of all ciphertext, tag pairs the user has received. For each new message a user sends, they would need to prove in zero-knowledge that their message is unique (or sufficiently dissimilar) with respect to the set *or* it is constructed as a forward of an appropriate message in the set. This is unlikely to be practical. In any case, both the trusted hardware and zero-knowledge solutions are susceptible to similarity-evasion attacks in which the adversary specially crafts a message that evades similarity detection but continues to carry the malicious intent.

Mitigating abuse of abuse mitigations. Message tracing can be used to identify the source of malicious or harassing messages. But the same techniques can be used against whistleblowers or activists. Robust policy dictating how and when to perform tracing is necessary for protection of users' privacy expectations. Our proposed schemes offer one such policy; a single report from a user that has received a message unlocks the ability for the platform to trace. Of course, higher level policies within the platform can dictate if a report is acted on, but this leaves open an interesting question if it is possible to integrate more expressive reporting and tracing policies directly into the cryptographic tracing scheme.

One interesting approach is to couple a tracing mechanism with an anonymous blacklisting scheme [18]. In anonymous blacklisting, cryptographic material bound to each message can be used to bar the author from further participation without linking messages together or identifying the author. One could extend this to tracing schemes, allowing the platform the ban the source or forwarders of a message without learning their identities. To be meaningful, this would require a tracing scheme that did not depend on metadata.

A second interesting approach for tree traceback is threshold reporting, where a message would need to be reported by many users before it can be traced. Once a certain threshold of reports is reached, some part or all of the trace tree could be revealed. Care would need to be taken to prevent one malicious client from hitting some reporting threshold q by forwarding the message to q colluding clients. And there are questions of the appropriate threshold and if it should or could vary with the size of the forwarding tree.

Finally, one could imagine various mechanisms for increasing the robustness of the tracing authority. We could require multiple parties to cooperate to recover a trace. Or delegating tracing authority to a separate party. Similarly we could imagine the tracing authority operating as part of a contractual anonymity system [27], in which a verifiable third-party mediates between users and the authority, only giving the authority the ability to de-anonymize if a user breaks their contract policy.

9 RELATED WORK

Tracing mechanisms. To the best of our knowledge, there are no works directly addressing tracing of forwards in E2E encrypted messaging or messaging in general. A number of works, including [9, 15, 20] have considered the problem of tracing payments in electronic cash systems. This is a conceptually related problem. However, the techniques are not directly applicable. First, these systems do not deal with binding the content of a message to the trace. Second most systems assume far more interaction with a central party (e.g., a bank) than is allowed in our setting.

Another line of work [25, 29, 30] considered IP packet traceback. But the approaches taken here are probabilistic. The point is not to trace an individual packet, but to get a trace of a stream of packets. As such packets are probabilistically marked [25] or kept in a Bloom filter [29]. They cannot be used to reliably trace an individual message.

A recent work [28] takes a different approach, supporting privacy-preserving path validation for a single packet. However, the approach requires the path to be known in advance so the appropriate session keys can be generated. It does not work for the ad-hoc paths that arise as messages are forwarded through a social network.

Secure logging. A sequence of works, including [26] considered secure logging mechanisms. The goal is to record on a local machine a log of all activity that both cannot be modified by an attacker and does not reveal anything without access to a decryption key. Typically such systems assume a third party who holds a key that is used to authenticate logs via signatures or hashes. Our tracing approach is, in a sense, a form of secure log in that we want (1) confidentiality from the party who stores the log (the messaging platform) until a key is revealed and (2) integrity from the party creating the log (the communicating users). However, our "log" has a richer structure we must obscure, more specific integrity guarantees, and the setting requires us to split the roles for preserving integrity and confidentiality.

Symmetric searchable encryption. A long line of symmetric searchable encryption (SSE) schemes use "encrypted pointers" similar to ours to hide the structure of an index held on an untrusted server. In particular we use techniques reminiscent of [10, 12] to construct our tracing mechanism extended to bind message plaintexts into the pointers and protect against maliciously generated pointers. Our functionality, confidentiality, and integrity goals are, however, distinct.

Traitor tracing. Another line of work explores traitor tracing mechanisms [7, 11] for identifying who within a group leaked a particular piece of content or a key to an outside party. These schemes are not directly applicable for a few reasons. First, the goals of message tracing are not to trace who leaked/reported the message to the platform (out of the forwarding chain "group"). Second, message tracing has very specific restrictions on who is able to perform tracing. For example, by user trace confidentiality, even members of the forwarding chain "group" should not be able to learn the message trace.

Signatures and aggregate signatures. Aggregate signatures were first proposed by Boneh et al. [8] in part to allow the path of a message through a network to be authenticated by an aggregation of the per-hop signatures and a list of the hops taken. In contrast, our scheme achieves its results without requiring the message to contain a linear list of its path-to-date and without disclosing that path to intermediaries.

Message franking. Message franking [13, 17] is another line of work that aims to prevent abuse in end-to-end encrypted messaging. While these protocols inspire our work, they cannot be used to report messages past one hop: a forwarded message can only be attributed to the last forwarder. One might attempt to employ message franking to trace further by sequentially interacting with each previous sender requiring them to either provide the message franking opening for the previous message (if the message was a forward) or be identified as the original source for the message. Such a solution would be impractical in the face of offline users and large forwarding trees.

Automated moderation systems. Various works have explored ad-hoc moderation[16] or machine learning to detect abusive content [24]. We do not provide a exhaustive list here. These works are promising but require the originators of messages be identified if they are to be held accountable or prevented from sending further messages.

10 CONCLUSION

We introduced tracing for E2E encrypted messaging. In this setting, a messaging platform can recover and cryptographically verify the path a message took as it was forwarded between users given a report by one of the recipients. We gave two schemes for tracing messages with different traceback targets: path to message source and entire message forwarding tree. Implementation benchmarks show both schemes are efficient and require the messaging platform to store less than 100 bytes of additional data per message sent.

ACKNOWLEDGMENTS

This work was supported in part by NSF awards DGE-1650441 and CNS-1704527.

REFERENCES

[1] 2009. Redis. https://redis.io/

[2] 2013. Signal. https://signal.org/

[3] 2013. Telegram. https://telegram.org/

[4] 2016. Rust Crypto. https://github.com/RustCrypto

[5] 2016. WhatsApp. https://www.whatsapp.com/

[6] Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *EUROCRYPT.*

[7] Dan Boneh and Matthew K. Franklin. 1999. An Efficient Public Key Traitor Tracing Scheme. In *CRYPTO.*

[8] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and Verifiably Encrypted Signatures from Bilinear Maps. In *EUROCRYPT.*

[9] Jan Camenisch, Ueli M. Maurer, and Markus Stadler. 1997. Digital Payment Systems With Passive Anonymity-Revoking Trustees. *Journal of Computer Security* (1997).

[10] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. 2013. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *CRYPTO.*

[11] Benny Chor, Amos Fiat, and Moni Naor. 1994. Tracing Traitors. In *CRYPTO.*

[12] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. 2006. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS.* ACM.

[13] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast Message Franking: From Invisible Salamanders to Encryptment. In *CRYPTO.*

[14] Facebook. 2017. Messenger Secret Conversations technical whitepaper. https://fbnewsroomus.files.wordpress.com/2016/07/messenger-secret-conversations-technical-whitepaper.pdf.

[15] Yair Frankel, Yiannis Tsiounis, and Moti Yung. 1996. "Indirect Discourse Proof": Achieving Efficient Fair Off-Line E-cash. In *ASIACRYPT.*

[16] R Stuart Geiger. 2016. Bot-based collective blocklists in Twitter: the counterpublic moderation of harassment in a networked public space. *Information, Communication & Society* 19, 6 (2016), 787–803.

[17] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. 2017. Message Franking via Committing Authenticated Encryption. In *CRYPTO.*

[18] Ryan Henry and Ian Goldberg. 2013. Thinking inside the BLAC box: smarter protocols for faster anonymous blacklisting. In *WPES.* ACM.

[19] Mike Isaac and Kevin Roose. 2018. Disinformation Spreads on WhatsApp Ahead of Brazilian Election. https://www.nytimes.com/2018/10/19/technology/whatsapp-brazil-presidential-election.html

[20] Dennis Kügler and Holger Vogt. 2002. Offline Payments with Auditable Tracing. In *Financial Cryptography (FC).*

[21] Joshua Lund. 2018. Technology preview: sealed sender for Signal. https://signal.org/blog/sealed-sender/

[22] Alexis Madrigal. 2018. India's Lynching Epidemic and the Problem With Blaming Tech. https://www.theatlantic.com/technology/archive/2018/09/whatsapp/571276/

[23] Farhad Manjoo. 2018. The Problem With Fixing WhatsApp? Human Nature Might Get in the Way. https://www.nytimes.com/2018/10/24/technology/fixing-whatsapp-disinformation-human-nature.html

[24] Benjamin Mullin. 2017. The New York Times is teaming up with Alphabet's Jigsaw to expand its comments. https://www.poynter.org/news/new-york-times-teaming-alphabets-jigsaw-expand-its-comments

[25] Stefan Savage, David Wetherall, Anna R. Karlin, and Thomas E. Anderson. 2000. Practical network support for IP traceback. In *SIGCOMM.*

[26] Bruce Schneier and John Kelsey. 1999. Secure Audit Logs to Support Computer Forensics. *ACM Trans. Inf. Syst. Secur.* 2, 2 (1999), 159–176.

[27] Edward J. Schwartz, David Brumley, and Jonathan M. McCune. 2010. Contractual Anonymity. In *NDSS.* The Internet Society.

[28] Binanda Sengupta, Yingjiu Li, Kai Bu, and Robert H. Deng. 2019. Privacy-Preserving Network Path Validation. Cryptology ePrint Archive, Report 2019/407. https://eprint.iacr.org/2019/407.

[29] Alex C. Snoeren. 2001. Hash-based IP traceback. In *SIGCOMM.*

[30] Dawn Xiaodong Song and Adrian Perrig. 2001. Advanced and Authenticated Marking Schemes for IP Traceback. In *INFOCOM.*

[31] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *SOSP.* ACM.

[32] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: scalable private messaging resistant to traffic analysis. In *SOSP.* ACM.

[33] WhatsApp. [n. d.]. https://faq.whatsapp.com/en/android/26000165/

[34] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in Numbers: Making Strong Anonymity Scale. In *OSDI.* USENIX.

```
$\text{RoR-CPA}^{\mathcal{A},b}_{\mathcal{E},m}$:
  $(K_1,\ldots,K_m) \leftarrow\$ \text{KeyGen}^m$
  $b' \leftarrow\$ \mathcal{A}^{\text{Fn}}$
  return $b'$

$\text{Fn}(i, M)$:
  $C_1 \leftarrow\$ \text{Enc}_{K_i}(M)$
  $C_0 \leftarrow\$ \{0,1\}^{\text{clen}(|M|)}$
  return $C_b$
```

Figure 11: Multi-key variant of Real-or-random chosen plaintext security game.

```
$\text{PRF}^{\mathcal{A},1}_{F,m}$:
  $(K_1,\ldots,K_m) \leftarrow\$ \text{KeyGen}^m$
  $b' \leftarrow\$ \mathcal{A}^{\text{Fn}}$
  return $b'$

$\text{Fn}(i, X)$:
  return $F_{K_i}(X)$
```

```
$\text{PRF}^{\mathcal{A},0}_{F,m}$:
  $(\rho_1,\ldots\rho_m) \leftarrow\$ \text{Func}(n,n)^m$
  $b' \leftarrow\$ \mathcal{A}^{\text{Fn}}$
  return $b'$

$\text{Fn}(i, X)$:
  return $\rho_i(X)$
```

Figure 12: Multi-key variant of PRF security game.

A CONFIDENTIALITY PROOF

First we recall the standard security notions needed in our proof. The real-or-random chosen plaintext attack (RoR-CPA) extended for multiple keys is defined in Figure 11. The adversary is tasked with determining whether ciphertexts are generated through the encryption algorithm or as a random bit string. The adversary's advantage is defined as

$$\mathbf{Adv}^{\text{ror-cpa}}_{\mathcal{E},m}(\mathcal{A}) = \left|\Pr\left[\text{RoR-CPA}^{\mathcal{A},1}_{\mathcal{E},m} \Rightarrow \text{true}\right] - \Pr\left[\text{RoR-CPA}^{\mathcal{A},0}_{\mathcal{E},m} \Rightarrow \text{true}\right]\right| .$$

The PRF security game extended for multiple keys is defined in Figure 12. The adversary is tasked with determining whether it is interacting with the PRF keyed on key i or a random function ρ_i. The adversary's advantage is defined as

$$\mathbf{Adv}^{\text{prf}}_{F,m}(\mathcal{A}) = \left|\Pr\left[\text{PRF}^{\mathcal{A},1}_{F,m} \Rightarrow \text{true}\right] - \Pr\left[\text{PRF}^{\mathcal{A},0}_{F,m} \Rightarrow \text{true}\right]\right| .$$

Theorem 1. *Let* MT *be the message tracing scheme for path traceback defined in Figure 3 using hash function H. Then if H is modeled as a random oracle, for any* PTrCONF *adversary $\mathcal{A}$ that makes at most q oracle queries, we give adversary $\mathcal{B}$ and C such that*

$$\mathbf{Adv}^{p\text{-}tr\text{-}conf}_{\text{MT}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F,q}(\mathcal{B}) + \mathbf{Adv}^{ror\text{-}cpa}_{\mathcal{E},q}(C)$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ and C run in time $T' \approx T$ and $\mathcal{B}$ and C make at most q oracle queries.

Proof. We bound the advantage of $\mathcal{A}$ in the PTrCONF game by bounding the advantage of each of a series of game hops, shown in Figure 13. At a high level, there are two main transitions. The

```
$G0$ $\boxed{G1}$ :
$b' \leftarrow\!\$\ \mathcal{A}^{\mathrm{Chal}}$
return $b'$

$\mathrm{Chal}(U_1, U_2, k_{i-1}, p)$:
$k_i \leftarrow\!\$\ \{0,1\}^n$
if $k_i \notin \mathsf{K}$ then
    $\mathsf{K}[k_i] \leftarrow j;\ j \leftarrow j+1$
$\widetilde{k}_i \leftarrow H(k_i)$
$mid \leftarrow F_{k_i}(p)$; $\boxed{mid \leftarrow \rho_{\mathsf{K}[k_i]}(p)}$
$ct \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(k_{i-1})$
$tt_s \leftarrow (mid, ct)$
return $tt_s$
```

```
$G2$ $\boxed{G3}$ :
$b' \leftarrow\!\$\ \mathcal{A}^{\mathrm{Chal}}$
return $b'$

$\mathrm{Chal}(U_1, U_2, k_{i-1}, p)$:
$k_i \leftarrow\!\$\ \{0,1\}^n$
if $k_i \notin \mathsf{K}$ then
    $\mathsf{K}[k_i] \leftarrow j;\ j \leftarrow j+1$
$\widetilde{k}_i \leftarrow H(k_i)$
$mid \leftarrow \{0,1\}^n$
$ct \leftarrow \mathsf{Enc}_{\widetilde{k}_i}(k_{i-1})$; $\boxed{ct \leftarrow\!\$\ \{0,1\}^{\mathrm{clen}(n)}}$
$tt_s \leftarrow (mid, ct)$
return $tt_s$
```

Figure 13: Security game hops for platform trace confidentiality proof for path traceback.

first transition replaces the PRF evaluation with an evaluation of a random function. The distinguishing advantage is bound by the PRF security of F. The second transition samples ct randomly instead of setting it as the output of an encryption algorithm. We bound the distinguishing advantage by RoR-CPA advantage against the symmetric encryption scheme.

G0 is the same as $\mathrm{PTrCONF}^{\mathcal{A},1}$ with TagGen unrolled and a bookkeeping table K added. The table K stores the tracing keys sampled in the challenge oracle by an index j. G1 is the same as G0 except mid is set as the output of one of a set of random functions ρ indexed by the key index $\mathsf{K}[k]$. Consider a multi-key PRF adversary $\mathcal{B}$ that replaces the PRF call setting mid with a call to its oracle (with a new index for each query). $\mathcal{B}$ runs exactly G0 and G1 for challenge bit $b = 1$ and $b = 0$ respectively,

$$|\Pr[\,\mathrm{G0} \Rightarrow \mathsf{true}\,] - \Pr[\,\mathrm{G1} \Rightarrow \mathsf{true}\,]| = \mathbf{Adv}^{\mathrm{prf}}_{F,q}(\mathcal{B})\ .$$

G2 replaces the call to the random function by simply randomly sampling mid. The difference between these two ways of sampling mid is that if the same key k was sampled twice on the same plaintext p, then G1 would make a duplicate call to $\rho_{\mathsf{K}[k]}(p)$ and mid would be the same. However, the probability that this occurs is the exact same as if a random sample of mid is a duplicate. Thus,

$$\Pr[\,\mathrm{G1} \Rightarrow \mathsf{true}\,] = \Pr[\,\mathrm{G2} \Rightarrow \mathsf{true}\,]\ .$$

Lastly, G3 is the same as G2 except ct is set as a random value. Consider a multi-key RoR-CPA adversary C that replaces the encryption call with a call to its encryption oracle. Then, C runs exactly G2 and G3 for challenge bit $b = 1$ and $b = 0$ respectively,

$$|\Pr[\,\mathrm{G2} \Rightarrow \mathsf{true}\,] - \Pr[\,\mathrm{G3} \Rightarrow \mathsf{true}\,]| = \mathbf{Adv}^{\mathrm{ror\text{-}cpa}}_{\mathcal{E},q}(C)\ .$$

Finally, note that the trace tag output in G3 is simply a fresh random value for each challenge oracle call, equivalent to $\mathrm{PTrCONF}^{\mathcal{A},0}$, completing the proof.

B ACCOUNTABILITY PROOF

First we recall the standard security notions needed in our proof. The multi-key variant for PRF security is defined in Figure 12 and described in the preliminaries of Appendix A. Collision resistance for PRFs (CR) is defined in Figure 14. The adversary is tasked with finding two different key-message pairs that evaluate to the same value under the PRF. The adversary's advantage is defined as

$$\mathbf{Adv}^{\mathrm{cr}}_{F}(\mathcal{A}) = \Pr\left[\,\mathrm{CR}^{\mathcal{A}}_{F} \Rightarrow \mathsf{true}\,\right]\ .$$

```
$\mathrm{CR}^{\mathcal{A}}_{F}$:
$((K, X), (K', X')) \leftarrow\!\$\ \mathcal{A}$
if $(K, X) = (K', X')$ then return false
return $F_K(X) = F_{K'}(X')$
```

Figure 14: PRF collision resistance security game.

Theorem 3. *Let* MT *be the message tracing scheme for path traceback defined in Figure 3. Then, for any TrUNF adversary $\mathcal{A}$ that makes at most q_{nm} new message queries, q_{s} send queries, and q_{sm} malicious send queries, we give adversaries $\mathcal{B}$ and C such that*

$$\mathbf{Adv}^{\textit{tr-unf}}_{\mathrm{MT},n}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F,q_{\mathrm{nm}}}(\mathcal{B}) + \mathbf{Adv}^{\textit{cr}}_{F}(C) + \frac{q_{\mathrm{s}} + q_{\mathrm{sm}}}{2^n}$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ and C run in time $T' \approx T$ and $\mathcal{B}$ makes at most $q_{\mathrm{s}} + q_{\mathrm{sm}}$ oracle queries.

Proof. This proof proceeds as a careful case analysis of the four adversary winning conditions. We will show that nearly all of the winning conditions correspond to an adversary finding a collision in the PRF F. In the single subcase that does not result in a collision, we will argue that the probability of reaching this subcase is low using the PRF security of F.

First, we move to a game which excludes the problematic winning condition, instead returning false, shown as a game hop from G0 to G1 as shown in Figure 15. The event of hitting the problematic winning condition is denoted by setting flag to bad. G0 is the same as $\mathrm{TrUNF}^{\mathcal{A}}$ with some extra bookkeeping added. First, the Svr-Trace algorithm is unrolled and the tracing keys used at each step of the traceback are saved into a table TrK. Second, in **Send**, tables S and SK are added to link the counter value s and k of the sent message to the identifier mid in PT. Third, a table, K is added to track the dummy tracing keys sampled in **NewMsg** and assigning them an index to be used in the multi-key PRF security game.

The early return of false in the problematic win condition is the relevant change for the hop from G0 to G1. Flag flag is set to bad in the main body if the WasFwd win condition is satisfied and the plaintext associated with $mid_{i,i+1}$ is the same as the reported plaintext. We will see in the case analysis why this specific win condition is problematic. In G1, instead of continuing executing after a flag is set, the game is aborted and false is returned. By the fundamental lemma of game playing [6],

$$\Pr[\,\mathrm{G0} \Rightarrow \mathsf{true}\,] \leq \Pr[\,\mathrm{G1} \Rightarrow \mathsf{true}\,] + \Pr[\,\mathrm{G0\ sets\ bad}\,]\ .$$

Next, consider G2 which is the same as G0 except during traceback, if the decrypted tracing key is in K, i.e. sampled in **Send**, then instead of executing PRF, one of a family of random functions ρ is executed. For simulation purposes, each of the random functions ρ_i are instantiated lazily. Note that this is the only place where the PRF is evaluated on a key sampled from **NewMsg**. Now consider the PRF adversary $\mathcal{B}$ that runs G2 replacing the call to ρ with a call to the PRF oracle and returning true if flag is set to bad. This corresponds exactly to the difference between G0 and G2 and thus,

$$\Pr[\,\text{G0 sets bad}\,] \le \Pr[\,\text{G2 sets bad}\,] + \mathbf{Adv}^{\text{prf}}_{F,m}(\mathcal{B})\ .$$

Next, we show through a case analysis that we can build an adversary C that can take a win from G1 and output a collision for F. The case analysis is for the four winning conditions of G1. We will return to bound the the probability of flag during discussion of the appropriate case.

- Case 1: $i^* \in [1, n] \wedge \tau = 1$

  By game construction, the reported tracing key, $k^* = k_{i^*,c^*}$, was populated following the output of a successful RecMsg in either **Send** or **SendMal**. A successful RecMsg implies the verification check $F_{k^*}(p^*) = mid_{c^*}$ succeeded. The call the RecMsg is made after mid_{c^*} is populated in PT with i^* as the recipient. Thus, during traceback the sender associated with $\mathsf{PT}[mid_{c^*}]$ will be added to the trace. Meaning the trace must include at least two users and this win condition is impossible.

- Case 2:

  $$\Big(j = 1 \wedge (\mathtt{WasSent}(tr_1, tr_2, mid_{1,2}, p^*) = \mathsf{false}\Big)$$

  For tr_1 to be added to the trace, we know $\mathsf{PT}[mid_{1,2}]$ is populated and has tr_1 as the sender. Given the oracles in our game, the only way to populate PT with $tr_1 \in [1, n]$ as sender is through the honest send oracle **SendMal** which is associated with an argument of counter $\mathsf{S}[mid_{1,2}] = s$, sending plaintext p_s. There are two subcases: (Case 2a) $p_s \neq p^*$, or (Case 2b) $p_s = p^*$. In both cases, we consider $k \leftarrow \mathsf{SK}[mid_{1,2}]$.

  - Case 2a: $p_s \neq p^*$
    We know from the execution of **Send** that $F_k(p_s) = mid_{1,2}$. We also know from the unrolled execution of Svr-Trace that $F_{trk_{1,2}}(p^*) = mid_{1,2}$. This gives us the collision, (k, p_s), $(trk_{1,2}, p^*)$.

  - Case 2b: $p_s = p^*$
    We will show that $trk_{1,2} \neq k$ leading to the collision $(k, p^*), (trk_{1,2}, p^*)$. Suppose $trk_{1,2} = k$. We know that from the execution of **Send**, the ciphertext $ct_{1,2}$ stored in $\mathsf{PT}[mid_{1,2}]$ is created as $\mathsf{Enc}_{H(k)}(k_{tr_1,s})$. We also know that since entries are never overwritten, that ciphertext remains.

    We know $k_{tr_1,s}$ is set, as that is a prerequisite of calling **Send**. Since

    $$\mathtt{WasSent}(tr_1, tr_2, mid_{1,2}, p^*) = \mathsf{false}\ ,$$

    $mid_s \neq \mathsf{auth}$ from the condition in **Send**. This implies mid_s and, by extension, $k_{tr_1,s}$ were not set in **NewMsg**. The only other place $k_{tr_1,s}$ can be set is following a successful RecMsg from either **Send** or **SendMal**. As in Case 1, this implies that the corresponding identifier in PT is populated with tr_1 as the sender, $\mathsf{PT}[F_{k_{tr_1,s}}(p^*)]$. Thus, during traceback, if $trk_{1,2} = k$, then $ct_{1,2}$ would be decrypted as $k_{tr_1,s}$ and the traceback would not stop at tr_1 since $\mathsf{PT}[F_{k_{tr_1,s}}(p^*)]$ is populated. All this means that $trk_{1,2} \neq k$ and the two keys collide on p^*.

```
G0 $\boxed{\text{G1}}$ G2 :
$c \leftarrow 0;\ r \leftarrow 0$
$(i^*, p^*, k^*, c^*) \leftarrow\!\!\$\ \mathcal{A}^{O}$
if $i^* \in [1, n]$ then $p^* \leftarrow p_{c^*};\ k^* \leftarrow k_{i^*,c^*}$
$j \leftarrow 0;\ k \leftarrow k^*$
$\mathsf{Tr}[j] \leftarrow i^*$
$mid \leftarrow F_k(p^*)$
while $mid \in \mathsf{PT}$ :
  $(ct, tr_s, tr_r) \leftarrow \mathsf{PT}[mid]$
  if $tr_r \neq \mathsf{Tr}[j]$ : break
  $\mathsf{Tr}[j+1] \leftarrow mid;\ \mathsf{Tr}[j+2] \leftarrow U_s;\ \mathsf{TrK}[j/2] \leftarrow k$
  $\widetilde{k} \leftarrow H(k);\ k \leftarrow \mathsf{Dec}_{\widetilde{k}}(ct)$
  $mid \leftarrow F_k(p^*)$
  if $k \in \mathsf{K}$ then $mid \leftarrow \rho_{\mathsf{K}[k]}(p^*)$
  $j \leftarrow j + 2$
$(tr_1, mid_{1,2}, tr_2, \ldots, mid_{\tau-1,\tau}, tr_\tau) \leftarrow \mathsf{Tr}^{-1}$
$(trk_{1,2}, \ldots, trk_{\tau-1,\tau}) \leftarrow \mathsf{TrK}^{-1}$
if $\big(i^* \in [1, n] \wedge \tau = 1\big)$ then return true
for $j \in 1$ to $\tau$ do
  if $tr_j \in [1, n]$ then
    if $\Big(j = 1 \wedge (\mathtt{WasSent}(tr_1, tr_2, mid_{1,2}, p^*) = \mathsf{false}\Big)$ then return true
    if $\Big(j = \tau \wedge \mathtt{WasRec}(tr_{\tau-1}, tr_\tau, mid_{\tau-1,\tau}, p^*) = \mathsf{false}\Big)$ then return true
    if $\Big(1 < j < \tau \wedge \mathtt{WasFwd}(tr_{i-1}, tr_i, mid_{i-1,i}, mid_{i,i+1}, p^*) = \mathsf{false}\Big)$ then
      $s \leftarrow \mathsf{S}[mid_{i,i+1}]$
      if $p_s = p^* \wedge \mathsf{SK}[mid_{i,i+1}] = trk_{i,i+1}$ then
        flag $\leftarrow$ bad; $\boxed{\textbf{return false}}$
      return true
return false

NewMsg($i$, $p$):
if $i \notin [1, n]$ then return $\bot$
$c \leftarrow c + 1$
$p_c \leftarrow p$
$k_{i,c} \leftarrow\!\!\$\ \mathsf{NewMsg}(i, p_c)$
if $k_{i,c} \notin \mathsf{K}$ then
  $\mathsf{K}[k_{i,c}] \leftarrow r;\ r \leftarrow r + 1$
$mid_c \leftarrow \mathsf{auth}$
return

Send($i$, $j$, $s$):
if $i \notin [1, n] \vee k_{i,s} = \bot$ then return $\bot$
$(k, tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}(i, j, p_s, k_{i,s})$
$((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Svr\text{-}Process}(\mathsf{PT}, i, j, tt_s)$
$\mathsf{PT}[mid] \leftarrow tt_p;\ \mathsf{S}[mid] \leftarrow s;\ \mathsf{SK}[mid] \leftarrow k$
if $mid_s = \mathsf{auth}$ then $\mathtt{WasSent}(i, j, mid, p_s) \leftarrow \mathsf{true}$
else $\mathtt{WasFwd}(i, j, mid_s, mid, p_s) \leftarrow \mathsf{true}$
if $j \in [1, n]$ then
  $c \leftarrow c + 1$
  $p_c \leftarrow p_s$
  $k_{j,c} \leftarrow \mathsf{RecMsg}(k, i, j, p_c, tt_r)$
  $\mathtt{WasRec}(i, j, mid, p_s) \leftarrow \mathsf{true}$
  $mid_c \leftarrow mid$
  return $tt_r$
return $(tt_r, k)$

SendMal($i$, $j$, $k$, $p$, $tt_s$):
if $i \in [1, n]$ then return $\bot$
$((mid, tt_p), tt_r) \leftarrow\!\!\$\ \mathsf{Svr\text{-}Process}(\mathsf{PT}, i, j, tt_s)$
$\mathsf{PT}[mid] \leftarrow tt_p$
if $j \in [1, n]$ then
  $k \leftarrow \mathsf{RecMsg}(k, p, tt_r)$
  if $k \neq \bot$ then
    $c \leftarrow c + 1$
    $p_c \leftarrow p$
    $k_{j,c} \leftarrow k$
    $\mathtt{WasRec}(i, j, mid, p) \leftarrow \mathsf{true}$
    $mid_c \leftarrow mid$
return $tt_r$
```

Figure 15: Game hop for path trace unforgeability.

- Case 3:

  $$\left(j = \tau \land \texttt{WasRec}(tr_{\tau-1}, tr_\tau, mid_{\tau-1,\tau}, p^*) = \mathsf{false}\right)$$

  To be last on a trace, tr_τ must be the reporting user i^*, reporting k_{i^*,c^*}. From the oracle structure, the tracing key, k_{i^*,c^*}, is set following a successful RecMsg in either **Send** or **SendMal**, at which point

  $$\texttt{WasRec}(U_{\tau-1}, tr_\tau, mid_{\tau-1,\tau}, p_{c^*}) \leftarrow \mathsf{true}$$

  for some $U_{\tau-1}$ where $p_{c^*} = p^*$. Since $mid_{\tau-1,\tau}$ is only set once in PT, the identities $U_{\tau-1} = tr_{\tau-1}$ and this case is impossible.
- Case 4:

$$\left(1 < j < \tau \land \texttt{WasFwd}(tr_{i-1}, tr_i, mid_{i-1,i}, mid_{i,i+1}, p^*) = \mathsf{false}\right)$$

Same as in Case 2, for tr_i to be added to the trace, we know $\mathsf{PT}[mid_{i,i+1}]$ is populated and has tr_i as the sender. Given the oracles in our game, the only way to populate PT with $tr_i \in [1, n]$ as sender is through the honest send oracle **SendMal** which is associated with an argument of counter $\mathsf{S}[mid_{i,i+1}] = s$, sending plaintext p_s. Again, there are two subcases: (Case 4a) $p_s \neq p^*$, or (Case 4b) $p_s = p^*$. In both cases, we consider $k \leftarrow \mathsf{SK}[mid_{i,i+1}]$.

- Case 4a: $p_s \neq p^*$
  Case 4a mirrors Case 2a. We know from the execution of **Send** that $F_k(p_s) = mid_{i,i+1}$. We also know from the unrolled execution of Svr-Trace that $F_{trk_{i,i+1}}(p^*) = mid_{i,i+1}$. This gives us the collision, $(k, p_s), (trk_{i,i+1}, p^*)$.
- Case 4b: $p_s = p^*$
  We will further consider two subsubcases. If $trk_{i,i+1} \neq k$, we have the trivial collision, (k, p^*), $(trk_{i,i+1}, p^*)$. Now consider the case $trk_{i,i+1} = k$. This subsubcase corresponds exactly to the problematic case set apart by flag_2. We know that from the execution of **Send**, $ct_{i,i+1} = \mathsf{Enc}_{H(k)}(k_{tr_i,s})$ is stored in $\mathsf{PT}[mid_{i,i+1}]$. Let us examine where $k_{tr_i,s}$ was set. Since

$$\texttt{WasFwd}(tr_{i-1}, tr_i, mid_{i-1,i}, mid_{i,i+1}, p_s) = \mathsf{false},$$

  then it would have been set in the else statement of **Send**, this implies the if statement was followed and therefore $mid_s = \mathsf{auth}$. This in turn means that $k_{tr_i,s}$ was set in **NewMsg**.

  Next, consider the unrolled traceback. Since $trk_{i,i+1} = k$, the ciphertext $ct_{i,i+1}$ will decrypt to $trk_{i-1,i} = k_{tr_i,s}$. Next, $F_{k_{tr_i,s}}(p^*) = mid_{i-1,i}$ leads to next step of traceback (also implying that $mid_{i-1,i} = mid_s$). This PRF evaluation would have been handled by a random function in G2. Thus, in G2, the probability flag_2 is set to bad is bounded by the probability that a call to **Send** or **SendMal** populated the mid corresponding to this random value,

  $$\Pr[\,\text{G2 sets bad}_2\,] \leq \frac{q_\text{s} + q_\text{sm}}{2^n}\ .$$

```
PTreeCONF$^{\mathcal{A},b}_{\text{MT}}$:
$b' \leftarrow\!\!\$\ \mathcal{A}^{\text{Chal}}$
return $b'$

NewClientState($U$, $mid$, $k$):
$gk \leftarrow\!\!\$\ \{0,1\}^n$
$tmd_{U,mid} \leftarrow (k, gk)$
$\mathsf{CTR}_U[tmd_{U,mid}] \leftarrow 0$

Chal($U_1$, $U_2$, $mid$, $p$):
if $tmd_{U_1,mid} = \bot$ then return $\bot$
$(k, tt^1_s) \leftarrow\!\!\$\ \mathsf{TagGen}(\mathsf{CTR}_{U_1}, U_1, U_2, p, tmd_{U_1,mid})$
$tt^0_s \leftarrow\!\!\$\ \{0,1\}^{\text{len}(tt_s)}$
return $tt^b_s$
```

```
UTreeCONF$^{\mathcal{A},b}_{\text{MT}}$:
$b' \leftarrow\!\!\$\ \mathcal{A}^{\text{Chal}}$
return $b'$

NewClientState($U$, $mid$, $k$):
$gk \leftarrow\!\!\$\ \{0,1\}^n$
$tmd_{U,mid} \leftarrow (k, gk)$
$\mathsf{CTR}_U[tmd_{U,mid}] \leftarrow 0$

Chal($st$, $U_1$, $U_2$, $mid$, $p$):
if $tmd_{U_1,mid} = \bot$ then return $\bot$
$tmd_0 \leftarrow tmd_{U_1,mid}$
$tmd_1 \leftarrow\!\!\$\ \mathsf{NewMsg}(U_1, p)$
$(k', tt_s) \leftarrow\!\!\$\ \mathsf{TagGen}(\mathsf{CTR}_{U_1}, U_1, U_2, p, tmd_b)$
$((mid, tt_p), tt'_r) \leftarrow \mathsf{Svr\text{-}Process}(st, U_1, U_2, tt_s)$
return $(k', tt'_r)$
```

Figure 16: Notions for (top) platform tree trace confidentiality and (bottom) user tree trace confidentiality.

This concludes the case analysis and gives us,

$$\Pr[\,\text{G1} \Rightarrow \mathsf{true}\,] \leq \mathbf{Adv}^{\text{cr}}_F(C)\ ,$$

completing the proof.

□

C CONFIDENTIALITY AND ACCOUNTABILITY FOR TREE TRACEBACK

Here we formalize security goals for tree traceback. We start with confidentiality notions which are essentially the same as those for path traceback. The main distinction is that clients in our tree traceback scheme have state (a set of counters). We add to the confidentiality notions from Section 5.1 the ability for the adversary to drive the honest client to an arbitrary state before submitting challenge queries. The confidentiality games for tree traceback are given in Figure 16. The unforgeability game remains largely the same, given in Figure 17. We add a recursive predicate check on all subtrees returned in the trace. Furthermore, the `WasRec` predicate is set even if a received message was not accepted. This captures the fact that a message will be traced as sent to a user regardless of whether the message was accepted.

C.1 Platform Trace Confidentiality

Theorem 4. *Let* MT *be the message tracing scheme for tree traceback defined in Figure 6 using hash function H. Then if H is modeled as a random oracle, for any* PTreeCONF *adversary $\mathcal{A}$ that makes at most q_{ns} new client state queries and q_{ch} challenge queries, we give adversary $\mathcal{B}$ and C such that*

$$\mathbf{Adv}_{\mathrm{MT}}^{p\text{-}tree\text{-}conf}(\mathcal{A}) \leq \mathbf{Adv}_{F,q_{\mathrm{ns}}+q_{\mathrm{ch}}}^{prf}(\mathcal{B}) + \mathbf{Adv}_{\mathcal{E},q_{\mathrm{ch}}}^{ror\text{-}cpa}(C) + \frac{q_{\mathrm{ns}}^2}{2^n}$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ and C run in time $T' \approx T$ and make at most $3q_{\mathrm{ch}}$ *oracle queries.*

Proof sketch: The proof proceeds with the same strategy as for path traceback. The first transition replaces the PRF executions with executions of a random function, where we can bound the distinguishing advantage by the PRF security of F since the PRF keys are random and not revealed to the adversary. There is a subtlety here that since the generator can be reused across multiple challenge oracle calls (with an incremented counter), it can be distinguished from random if a generator is every resampled. Thus, we include a term bounding the low probability event of a generator resampling collision. The second transition replaces calls to the encryption algorithm with sampling random bits. After these transitions, the sender trace tag output from the challenge oracle is a *mid* which is the output of a random function, and three ciphertexts which are random bit strings.

C.2 User Trace Confidentiality

Theorem 5. *Let* MT *be the message tracing scheme for tree traceback defined in Figure 6. For any* UTreeCONF *adversary $\mathcal{A}$ that makes at most q_{ns} new client state queries and q_{ch} challenge queries, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\mathrm{MT}}^{p\text{-}tree\text{-}conf}(\mathcal{A}) \leq 2\mathbf{Adv}_{F,q_{\mathrm{ns}}+q_{\mathrm{ch}}}^{prf}(\mathcal{B}) + \frac{q_{\mathrm{ns}}^2}{2^{n-1}}$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ runs in time $T' \approx T$ and makes at most $2q_{\mathrm{ch}}$ *oracle queries.*

Proof sketch: The adversary's view includes the recipient trace tag and the tracing key. The recipient trace tag is made up of a message identifier, sender key share ciphertext, and platform key share. The tracing key is dependent on the challenge bit chosen generator, and so by extension the message identifier and sender key share ciphertext are as well, since they are derived as functions of the tracing key. We use the PRF security of F to transition to an intermediate game where the tracing key is derived from a random function. Similar to as in the sketch above for platform trace confidentiality, since generators are reused across calls to the challenge oracle (with an incremented counter), they can be distinguished if a generator is resampled. Thus, we include a term bounding the low probability event of a generator resampling collision.

```
TreeUNF$^{\mathcal{A}}_{\mathrm{MT},n}$:
$c \leftarrow 0$
$(i^*, p^*, tmd^*, c^*) \leftarrow\!\!\$\ \mathcal{A}^{O}$
if $i^* \in [1, n]$ then $p^* \leftarrow p_{c^*}$; $tmd^* \leftarrow tmd_{i^*,c^*}$
$tr_0, clist_0 \leftarrow$ Svr-Trace(PT, $i^*, p^*, tmd^*$)
for $(mid_{0,1}, tr_1, clist_1) \in clist_0$ do
  if $(tr_0 \in [1, n] \wedge$ WasSent$(tr_0, tr_1, mid_{0,1}, p^*) =$ false$)$ then return true
  if check_tree$(tr_0, mid_{0,1}, tr_1, clist_1)$ then return true
return false

check_tree$(tr)$:
$(tr_{i-1}, mid_{i-1,i}, tr_i, clist_i) \leftarrow tr$
if $clist_i = \emptyset$ then
  if $(tr_i \in [1, n] \wedge$ WasRec$(tr_{i-1}, tr_i, mid_i, p^*) =$ false$)$ then return true
for $(mid_{i,i+1}, tr_{i+1}, clist_{i+1}) \in clist_i$ do
  if $(tr_i \in [1, n] \wedge$ WasFwd$(tr_i, tr_{i+1}, mid_{i-1,i}, mid_{i,i+1}, p^*) =$ false$)$ then return true
  if check_tree$(tr_i, mid_{i,i+1}, tr_{i+1}, clist_{i+1})$ then return true
return false

NewMsg$(i, p)$:
if $i \notin [1, n]$ then return $\perp$
$c \leftarrow c + 1$
$p_c \leftarrow p$
$tmd_{i,c} \leftarrow\!\!\$$ NewMsg$(i, p_c)$
$mid_c \leftarrow$ auth
return

Send$(i, j, s)$:
if $i \notin [1, n] \vee tmd_{i,s} = \perp$ then return $\perp$
$(k, tt_s) \leftarrow\!\!\$$ TagGen(CTR$_i$, $i, j, p_s, tmd_{i,s}$)
$((mid, tt_p), tt_r) \leftarrow\!\!\$$ Svr-Process(PT, $i, j, tt_s$)
PT$[mid] \leftarrow tt_p$
if $mid_s =$ auth then WasSent$(i, j, mid, p_s) \leftarrow$ true
else WasFwd$(i, j, mid_s, mid, p_s) \leftarrow$ true
if $j \in [1, n]$ then
  $c \leftarrow c + 1$
  $p_c \leftarrow p_s$
  $tmd_{j,c} \leftarrow$ RecMsg$(k, i, j, p_c, tt_r)$
  WasRec$(i, j, mid, p_s) \leftarrow$ true
  $mid_c \leftarrow mid$
  return $tt_r$
return $(tt_r, k)$

SendMal$(i, j, k, p, tt_s)$:
if $i \in [1, n]$ then return $\perp$
$((mid, tt_p), tt_r) \leftarrow\!\!\$$ Svr-Process(PT, $i, j, tt_s$)
PT$[mid] \leftarrow tt_p$
if $j \in [1, n]$ then
  WasRec$(i, j, mid, p) \leftarrow$ true
  $tmd \leftarrow$ RecMsg$(k, p, tt_r)$
  if $tmd \neq \perp$ then
    $c \leftarrow c + 1$
    $p_c \leftarrow p$
    $tmd_{j,c} \leftarrow tmd$
    $mid_c \leftarrow mid$
return $tt_r$
```

Figure 17: Trace unforgeability security game for tree traceback.

C.3 Trace Unforgeability

Theorem 6. *Let* MT *be the message tracing scheme for tree traceback defined in Figure 6. Then, for any TreeUNF adversary $\mathcal{A}$ that makes at most q_{nm} new message queries, q_{s} send queries, and q_{sm} malicious send queries, we give adversary $\mathcal{B}$ such that*

$$\mathbf{Adv}_{\mathrm{MT},n}^{tree\text{-}unf}(\mathcal{A}) \leq \mathbf{Adv}_{F}^{cr}(\mathcal{B}) + \frac{(q_{\mathrm{nm}} + q_{\mathrm{s}} + q_{\mathrm{sm}})^2}{2^n}$$

where if $\mathcal{A}$ runs in time T, then $\mathcal{B}$ runs in time $T' \approx T$.

Proof sketch: The proof strategy is the same as in trace unforgeability for path traceback. We will perform a case analysis of the three winning conditions of the security game and show that in each one, we will either be able to show a collision of F or can show that it is a low probability event by the PRF security of F.

- Case 1: honest root did not author message

  $$\texttt{WasSent}(tr_0, tr_1, mid_{0,1}, p^*) = \mathsf{false}$$

  For $mid_{0,1}$ to be added to the trace, we know $\mathsf{PT}[mid_{0,1}]$ is populated and has tr_0 as the sender. Given the oracles in our game, the only way to populate PT with $tr_0 \in [1, n]$ as sender is through the honest send oracle **SendMal** which was called with a message id counter s indicating sending plaintext p_s. There are two subcases: (Case 2a) $p_s \neq p^*$, or (Case 2b) $p_s = p^*$.

  - Case 1a: $p_s \neq p^*$
    We consider the tracing key generated during tag generation in the oracle call; call this $k_{0,1}$. We will also consider the tracing key used in traceback to include $mid_{0,1}$ in the trace, call this $trk_{0,1}$. We know from the execution of **Send** that $F_{k_{0,1}}(p_s) = mid_{0,1}$. We also know from the execution of Svr-Trace that for $mid_{0,1}$ to have been included in the trace, there must have been a tracing key $trk_{0,1}$ used such that $F_{trk_{0,1}}(p^*) = mid_{0,1}$. This constitutes a collision.
  - Case 1b: $p_s = p^*$
    For tr_0 to have been identified as the root of the trace, there must have been some mid_b that had tr_0 as the sender. The root identifying condition is either that the escrowed tracing key evaluates to an invalid mid or that the current tracing key is not well-formed with respect to the escrowed generator. Since tr_0 is honest, we can rule out the second case, and the first case would only occur if tr_0 was the author of the message — since our oracles do not make it possible for an honest user to forward a message that was not received from the server and accepted by RecMsg. So tr_0 is the author of a s' counter associated with p^*. Any call to **Send** of s' would result in the WasSent predicate being sent to true since s' was authored. Thus, $s \neq s'$. This means that either the generator for s and the generator for s' collide on a tracing key, the tracing key from s and the tracing key from s' collide on $mid_{0,1}$, or the generators are the same for s and s'. The last case we can bound as a low probability event.

- Case 2: honest leaf did not receive message

  $$\texttt{WasRec}(tr_{i-1}, tr_i, mid_{i-1,i}, p^*) = \mathsf{false}$$

  To be added to the trace, it must be that $mid_{i-1,i}$ in PT contained tr_{i-1} as sender and tr_i as the recipient. From our oracle construction, the only way for a pair of users to be added to PT is through the **Send** or **SendMal** oracles, which both set the WasRec predicate with plaintext p input to the oracle. In which case, if $p^* = p$, the WasRec would have been true. Therefore, $p^* \neq p$ and there exists a collision with the tracing key used in traceback trk,p^* and the tracing key used in the oracle k,p on $mid_{i-1,i}$.

- Case 3: internal node did not forward message

  $$\texttt{WasFwd}(tr_i, tr_{i+1}, mid_{i-1,i}, mid_{i,i+1}, p^*) = \mathsf{false}$$

  Same as in Case 1, for $mid_{i,i+1}$ to be added to the trace, we know $\mathsf{PT}[mid_{i,i+1}]$ is populated and has tr_i as the sender. Given the oracles in our game, the only way to populate PT with $tr_i \in [1, n]$ as sender is through the honest send oracle **SendMal** which was called with a message id counter s indicating sending plaintext p_s. There are two subcases: (Case 2a) $p_s \neq p^*$, or (Case 2b) $p_s = p^*$.

  - Case 3a: $p_s \neq p^*$
    We consider the tracing key generated during tag generation in the oracle call; call this $k_{i,i+1}$. We will also consider the tracing key used in traceback to include $mid_{0,1}$ in the trace, call this $trk_{i,i+1}$. We know from the execution of **Send** that $F_{k_{i,i+1}}(p_s) = mid_{i,i+1}$. We also know from the execution of Svr-Trace that for $mid_{i,i+1}$ to have been included in the trace, there must have been a tracing key $trk_{i,i+1}$ used such that $F_{trk_{i,i+1}}(p^*) = mid_{i,i+1}$. This constitutes a collision.
  - Case 3b: $p_s = p^*$
    We will further consider two subsubcases. If $mid_{i-1,i} = mid_s$, then since the WasFwd was not set to true, we can infer that s was authored by tr_i and importantly, gk was sampled randomly. If $mid_{i-1,i} \neq mid_s$, then we can infer gk was the hashed output of the key shares provided for mid_s, which importantly, likely differ from the key shares provided by $mid_{i-1,i}$ created gk'. This means that there is a collision between gk,ctr and gk',ctr' to $k_{i,i+1}$, a collision between $k_{i,i+1}$,p^* and $trk_{i,i+1}$,p^* to $mid_{i,i+1}$, or gk and gk' are the same which we can bound to be low probability.