

Causal Awareness in End-to-End Encrypted Group Messaging

Joseph Jaeger¹ , Akshaya Kumar¹ , and Igors Stepanovs² 

¹ School of Cybersecurity and Privacy
Georgia Institute of Technology, Atlanta, Georgia, US
{josephjaeger, akshayakumar}@gatech.edu
<https://cc.gatech.edu/~josephjaeger/>
<https://cc.gatech.edu/~akumar805/>

² Amazon, Madrid, Spain
igoste@amazon.com

13 October 2026

Abstract. Chats do not always unfold in a single clear order. Two users may send messages before information about either message reaches the other sender. Such messages are concurrent, but most instant messengers provide no way for users to recognize this relationship. Chen and Fischlin (Eurocrypt 2024) studied this limitation in the two-user setting, showing that Signal does not expose enough information to recover the context in which a message was sent; they proposed a fix. We study group chats, where a user may need to reason about ordering and concurrency among other users’ messages. This can become a security problem when network delays or adversarial interference cause an authentic message to be shown in a misleading context.

Event ordering is a classical problem; Lamport clocks and vector clocks are standard starting points. Secure messaging needs a different treatment because communication is asynchronous, bandwidth is limited, and added latency is undesirable. To address the problem under these constraints, we introduce causal awareness: the ability of a receiver to reconstruct a specified part of the causal context in which an accepted message was sent. Senders transmit records of what they have sent and received. Receivers verify these records and compare them with their own.

We develop a framework of correctness and security notions for causal awareness in group chats. Our definitions are parameterized by how much causal information receivers must recover. We give baseline constructions and prove their properties. We also introduce checkpoint-aided and server-aided variants of the framework and prove that, in each, our notions of correctness and integrity imply causal awareness. We discuss how causal information could be surfaced in a messenger’s user interface.

Contents

1	Introduction	3
2	Preliminaries	6
3	Causality graphs	7
4	Group chats.....	9
4.1	Correctness	9
5	Unencrypted group chats	12
5.1	Graph correctness with respect to rGMax.....	13
5.2	Graph correctness with respect to rG2Hop.....	15
6	Encrypted group chats	16
6.1	Integrity of group chats	16
6.2	Causal awareness.....	17
6.3	Constructing causally aware group chats	18
7	Checkpoint-aided causal awareness	19
8	Server-aided causal awareness	21
9	Deployment costs and limitations	25
A	Proofs for group chat constructions	31
A.1	Proof of Theorem 1	31
A.2	Proof of Theorem 2	32
A.3	Proof of Theorem 4	33
A.4	Proof of Theorem 5	35
A.5	Proof of Theorem 6	36
A.6	Proof of Theorem 7	39
B	Checkpoint-aided proofs (Proof of Theorem 8)	41
C	Server-aided proofs (Proof of Theorem 9)	45
D	Related work	47

1 Introduction

Effective communication relies on messages being read in the context in which they were sent. In messaging applications, this context may differ across users. Consider the conversation in Fig. 1, where Alice, Bob, and Charlie are planning a meetup. No message was forged or modified, yet Bob and Charlie walk away with different conclusions about what was agreed. This condition, where different users hold different views of the same conversation, is what we call *causality confusion*.

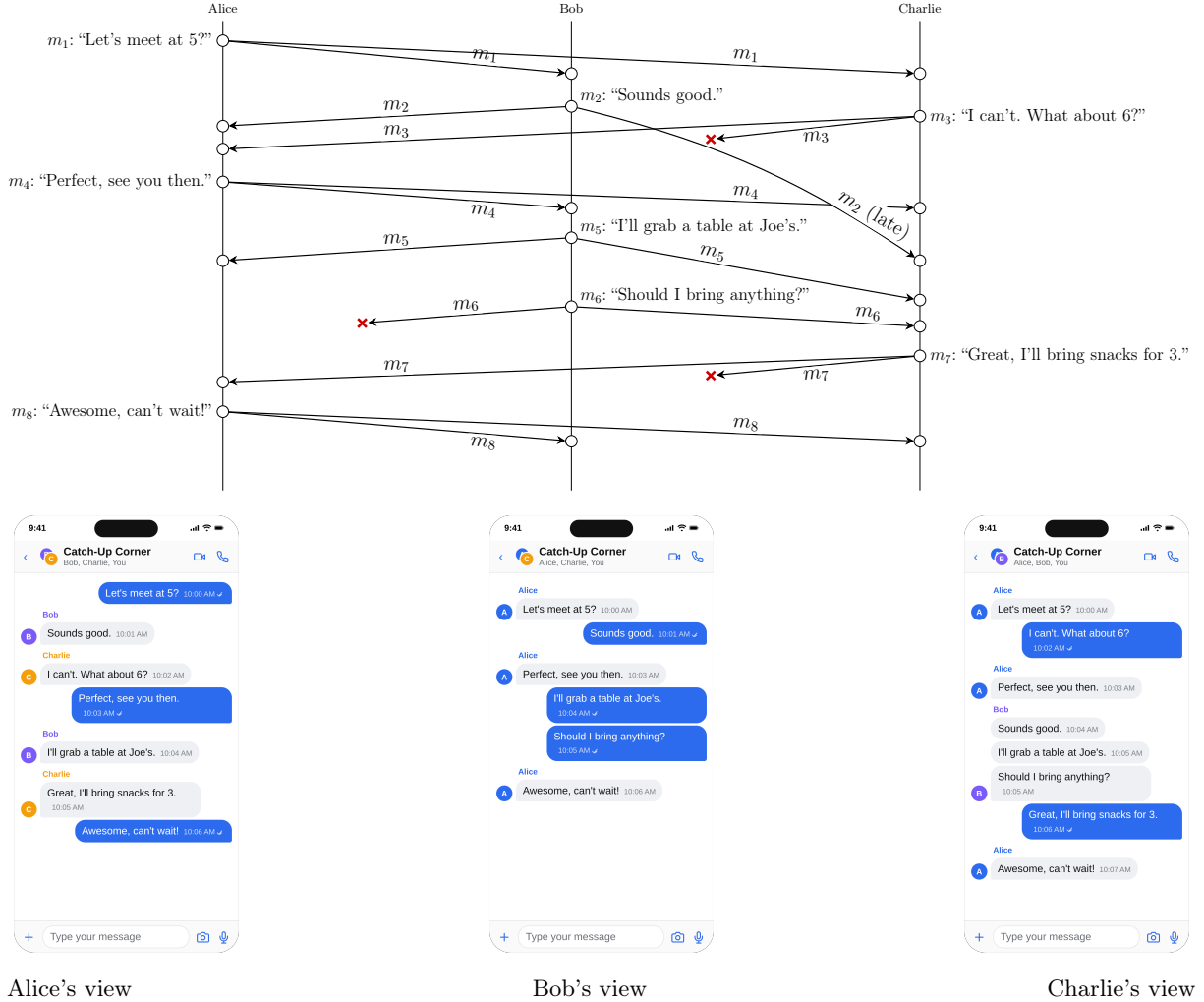


Fig. 1: An example of causality confusion in group chats.

We address causality confusion in end-to-end encrypted group chats. Deployed secure messengers and their group variants have been studied extensively [UDB⁺15, CCD⁺20, ACD19, CCG⁺18, RMS18, JK25]. They protect message content by ensuring that attackers cannot learn messages or inject messages into the chat (confidentiality and integrity). Their ordering guarantees, however, are limited. A receiver can recover the order in which any single sender sent their messages, but the protocol makes no guarantee about how messages from different senders interleave. This means that causally unrelated messages from different senders may be delivered to different users in different orders. Here, concurrency is causal rather than wall-clock concurrency: two users need not send at the same instant. It may arise from network

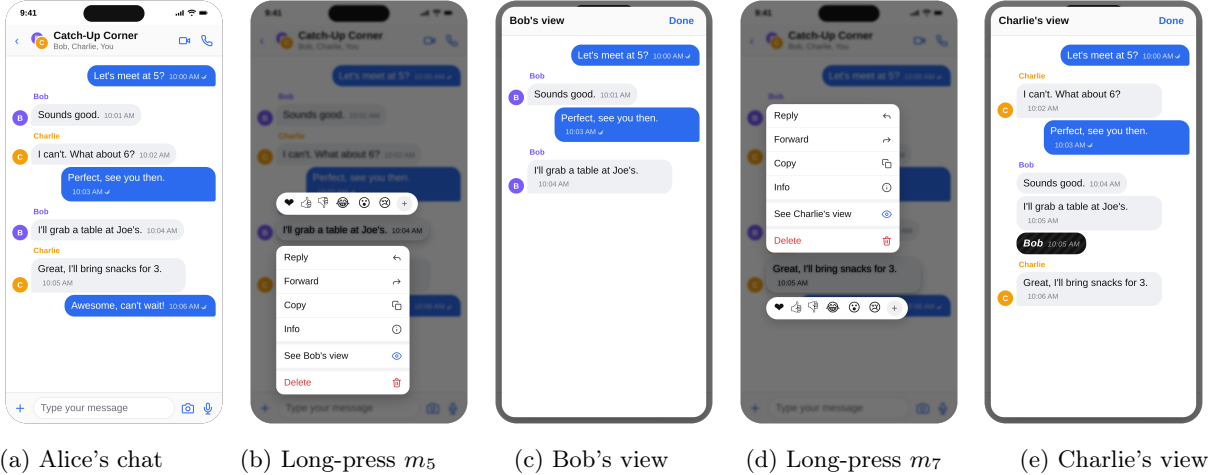


Fig. 2: The proposed UI feature for causal awareness, walked through from Alice’s point of view. (a) Alice’s chat as she sees it in our prototype, for the same conversation as in Fig. 1 (bottom row, leftmost). (b) She long-presses Bob’s “I’ll grab a table at Joe’s.” (m_5); the action menu surfaces *See Bob’s view*, which shows (c) Bob’s view of the conversation at the moment he sent m_5 . (d) Alice long-presses Charlie’s “Great, I’ll bring snacks for 3.” (m_7); the menu now offers *See Charlie’s view*, which shows (e) Charlie’s view at m_7 . The black placeholder in (e) stands for Bob’s m_6 , which Charlie received before sending m_7 , but Alice has not yet received. Alice thus learns that Charlie had seen a message she has not, but she does not learn its contents until she receives it herself.

latency or active meddling by the server or network adversary. Concurrency and message loss lead to causality confusion.

One way to avoid causality confusion would be to prevent users from having different views altogether, i.e., require consistency of views. However, achieving consistency is expensive. It requires either extra rounds of communication between users to reach consensus on message order, trusting the delivery service to impose a total order, or blocking a user from displaying a message until all its causal predecessors have arrived [AAN⁺24, AAN⁺22, AMT23, WKHB21]. None of these is acceptable in asynchronous end-to-end encrypted messaging. This mirrors fork-consistency results for untrusted storage, where preventing split views requires synchronization under concurrency [CsS07].

A better-suited goal for messengers is to ensure that users are aware of the context in which each message is sent. We call this *causal awareness*. It requires the protocol to give each user enough information to locally reconstruct the relevant part of the global transcript, allowing them to learn the views of other users. In Fig. 2 we illustrate what such a guarantee would look like in the user interface of a messaging application. We revisit the causality confusion example from before: Alice can now long-press a message from Bob or Charlie and see exactly what their view of the conversation was when they sent it. This interface is only one possible way to expose the cryptographic guarantee. Prior work on usable security shows that users often do not invoke or interpret security UI as designers expect [WT99, FKBS25], so our formal definitions do not rely on users manually checking every message.

In this work, we develop a framework of correctness and security definitions for causal awareness in group chats and give constructions that achieve them. To model the causal dependencies of a conversation, we define and use causality graphs. Causality graphs, like the one shown in Fig. 1, are directed acyclic graphs (DAGs) whose vertices represent send and receive events, and edges represent message deliveries. Our causal awareness notion requires a receiver to reconstruct the relevant part of the global conversation graph locally. We parameterize the definition by what “relevant” means to capture a hierarchy of causal awareness notions. Our goal is to identify which notions support the intended user experience and what overhead they require.

The ordering of events is a classical problem in distributed systems and secure distributed protocols, going back to Lamport’s happened-before relation [Lam78] and work on securing causal relationships and group multicast [RG93,RB94]. Many settings have mature solutions, but their assumptions do not match end-to-end encrypted messaging: communication is asynchronous, users may be offline for long periods, message display should not block, and the delivery service is not trusted to impose the right order. Earlier secure-messaging work identified causality as a concern [UDB⁺15,RMS18,EMP18]; Chen and Fischlin [CF24] formalized causality preservation for two-party channels. Their definition requires a user to know all events preceding its latest action, i.e., the entire global graph up to that point. The strongest variant of our causal awareness notion extends their definition to the group setting. It can be achieved naively by having each user maintain their view of the conversation in a local causality graph and attach their entire local graph to every outgoing message. This solution is clearly impractical, but the cost is not only an artifact of the naive construction. This notion forces users to track messages between other users, even when they were not directly involved, so payload size grows with the group history.

We therefore introduce a weaker variant that only requires a receiver to have the causal information needed to reconstruct the sender’s view at the time of sending. To achieve this, each user records its send and receive actions and makes the relevant part of that record available with its outgoing messages. This property is exactly what is needed to realize the long-press feature in Fig. 2. The receiver can inspect what the sender had already seen, without learning unrelated parts of the group transcript.

In our analysis, we determine the minimal graph evidence required to satisfy these variants in the broadcast setting. We give constructions for both variants. We prove that the payloads of our construction for the strongest variant are optimal. We then augment a standard secure bidirectional channel with this information. Because our construction uses the secure channel as a black box, it preserves the channel’s cryptographic properties.

The alternative is to send the encrypted message normally and attach causal information as a separate authenticated object. This can reduce integration cost, but it has weaker availability. An attacker can deliver the message while dropping the causal component, creating a denial-of-service attack.

Even the two-hop variant can be expensive in the worst case. If some group member is offline for a long time, a sender may need to attach a growing history so that the offline member can eventually catch up. In large, long-lived groups, this overhead can become substantial. We therefore study two communication-saving refinements: checkpoint-aided causal awareness and server-aided causal awareness.

Checkpoint-aided causal awareness periodically reconciles causal information at explicit checkpoint events. This resembles epoch-based group-messaging designs such as MLS/TreeKEM, where Commit-like operations advance group state [WPBB23,WPB25]. Users can continue within an epoch, but must process checkpoint information to advance consistently. Checkpoints can reduce recurring per-message overhead by establishing a common baseline, but they leave the worst-case bound unchanged and introduce blocking at checkpoint boundaries. This may be acceptable in protocols where epoch transitions already have blocking semantics.

Server-aided causal awareness separates message delivery from causal-evidence delivery. A sender sends both a message ciphertext and causal evidence to the server; receivers get the message plus an authenticated pointer to the evidence. They can later fetch and verify the evidence when needed. This preserves non-blocking message display and moves causal evidence out of band, but a malicious server can withhold evidence and cause denial of service.

Our present treatment has three main limitations. We focus on static groups, outsider-style delivery attacks, and the interaction between causal awareness and standard channel integrity. We do not yet model malicious insiders, dynamic membership in full generality, or explicit post-compromise-security guarantees for the underlying messaging channel.

Related work. Closest to our notion of causal awareness is the causality preservation notion of Chen and Fischlin [CF24], who lift Lamport’s happened-before relation to a two-party causality graph and require a receiver to reconstruct the graph of all events preceding its latest receive. Our strongest variant (rGMax) extends their notion to the group setting. Because this extension is impractical, we introduce weaker, more efficient variants. Marson [Mar17] defines communication graphs and channels that record a user’s causal history and require it to match the true order of events, and Eugster, Marson, and Poetter-

ing [EMP18] build causal-broadcast channels. In contrast to our approach, both delay message delivery until predecessors arrive; neither considers message loss or out-of-order delivery. Related work otherwise falls into five categories: (i) causal ordering in distributed systems, from Lamport’s logical clocks [Lam78] and causal broadcast [BSS91,NMM18] to causally consistent data stores [LFKA11,DIRZ14,ASB18] and malicious-participant lower bounds [RG93,RB94]; (ii) ordering in two-party messaging [BCC⁺23,CF24] and its extension to message franking [CF24,NR25]; (iii) informal definitions of ordering and consistency in group messaging [UDB⁺15,GUVC09,Mar14,SH19,SVH18,LZPR24,JBHH21]; (iv) ordering in CGKA and MLS [BBR⁺23,BRO⁺25,AAN⁺22,AMT23,WKHB21,AAN⁺24,CsS07], none of which addresses application-message ordering; and (v) attacks on ordering in deployed messengers [SKH17,JK25,RMS18,PST23,AMPS26,GGG⁺26]. We give a more detailed discussion of related work in Appendix D.

Roadmap. The remainder of the paper is organized as follows. Section 3 defines causality graphs and their properties. Section 4 introduces the group-chat syntax that we use to capture causally aware group chats. It defines notions of message and graph correctness for group chats. In Section 5, we study unencrypted channels and identify what causal information each broadcast message must make available so that receivers can reconstruct the appropriate portion of the global transcript. It gives constructions that satisfy graph correctness. Section 6 introduces security notions for integrity and causal awareness. It proves that the causal awareness of a group chat follows from its graph correctness and integrity. It then presents the graph-then-encrypt construction, which composes an unencrypted graph-correct channel with an encrypted channel that provides plaintext and ciphertext integrity, thereby achieving causal awareness. Finally, Section 7 and Section 8 describe the checkpoint-aided and server-aided settings for causal awareness, giving associated security definitions and efficiency tradeoffs.

2 Preliminaries

Notation and game conventions. Let $\mathbb{N} = \{0, 1, 2, \dots\}$ and, for $n \geq 1$, let $[n] = \{1, \dots, n\}$. We use $\leftarrow$ and $\leftarrow^*$ for deterministic and randomized assignment, respectively; $(x, y) \leftarrow (a, b)$ simultaneously sets x to a and y to b . We let $\perp \notin \{0, 1\}^*$ be an error code that indicates rejection. Game variables are shared with their oracles; “require d ” returns $\perp$ if d is false. Even in an oracle body, **abort**(y) halts the entire enclosing game or adversary execution with output y ; proof games use **abort**(false) and reduction adversaries use **abort**($\perp$). Bracket-indexed pseudocode variables are tables whose unassigned entries read as $\perp$; tables therefore need no explicit initialization. We reserve $\emptyset$ for sets. In a figure specifying several games or variants, a right-hand comment marks a line as belonging only to the named games or variants; unannotated lines belong to all of them. We define $\max(\emptyset) = \perp$, and no element x satisfies $x \circ \perp$ for $\circ \in \{<, \leq, <_\ell\}$.

By $y \leftarrow^* \mathcal{A}^O(x)$ we denote the randomized execution of algorithm $\mathcal{A}$ on input x , with oracle access to O , producing output y . We generalize this to multiple inputs or oracles in the natural way. In reductions, SIMO answers the simulated adversary’s O queries, while O is an oracle of the reduction’s security game. If multiple oracles are provided, then $\mathcal{A}$ is allowed to call them arbitrarily many times, in any order. If $\mathcal{A}$ is deterministic, we use $\leftarrow$ instead. Running time is worst case. Adversaries are algorithms. We require that adversaries never pass $\perp$ as input to their oracles. We use the code-based game-playing framework of [BR06] for specifying security definitions and writing proofs. $\Pr[\mathcal{G}]$ denotes the probability that game $\mathcal{G}$ returns **true**. In security reductions, we omit specifying the running times of the constructed adversaries when they are roughly the same as the running time of the initial adversary. Unless stated otherwise, a displayed game $\mathcal{G}_\emptyset^x(\mathcal{A}_x)$ defining notion x uses the advantage convention

$$\text{Adv}_\emptyset^x(\mathcal{A}_x) := \Pr[\mathcal{G}_\emptyset^x(\mathcal{A}_x)].$$

In reduction figures, a comment $\mathcal{G}_{[i,j]}$ marks a line present only in games $\mathcal{G}_i, \dots, \mathcal{G}_{j-1}$, with $j = \infty$ denoting all remaining games.

Fundamental Lemma of Game Playing. In our proofs, we frequently make use of the Fundamental Lemma of Game Playing [BR06]. Suppose that games $\mathcal{G}_i$ and $\mathcal{G}_{i+1}$ are syntactically identical until a Boolean flag **bad** is set. Then

$$|\Pr[\mathcal{G}_i] - \Pr[\mathcal{G}_{i+1}]| \leq \Pr[\text{bad}^{\mathcal{G}_i}] = \Pr[\text{bad}^{\mathcal{G}_{i+1}}],$$

```

InitGraph( $n$ ) //  $n \in \mathbb{N}, n \geq 1$ 
 $V_i \leftarrow \{\}$   $\forall i \in [n]$ 
 $E \leftarrow \{\}$ ; Return  $(n, V_1, \dots, V_n, E)$ 

nextCtr( $V$ )
If  $|V| = 0$  then return 1 else return  $1 + \max\{v.\text{ctr} \mid v \in V\}$ 

AddSendVrtx( $\mathcal{G}, i, \text{eid}, \text{label}$ ) //  $i \in [n]$ 
 $(n, V_1, \dots, V_n, E) \leftarrow \mathcal{G}$ 
// Ignore duplicate send-event identifiers for  $i$ 
 $U \leftarrow \{v' \in V_i \mid v'.\text{type} = \text{SEND} \wedge v'.\text{eid} = \text{eid}\}$ 
If  $|U| \neq 0$  then return  $\mathcal{G}$ 
 $\text{ctr} \leftarrow \text{nextCtr}(V_i)$ ;  $v \leftarrow (i, \text{SEND}, \text{ctr}, \text{eid}, \text{label})$ 
 $V_i \leftarrow V_i \cup \{v\}$ ; Return  $(n, V_1, \dots, V_n, E)$ 

AddRecvVrtx( $\mathcal{G}, i, j, \text{eid}$ ) //  $i, j \in [n]$ 
 $(n, V_1, \dots, V_n, E) \leftarrow \mathcal{G}$ 
// Ignore missing or ambiguous send-event identifiers for  $i$ 
 $U \leftarrow \{v' \in V_i \mid v'.\text{type} = \text{SEND} \wedge v'.\text{eid} = \text{eid}\}$ 
If  $|U| \neq 1$  then return  $\mathcal{G}$ 
 $\{v'\} \leftarrow U$ ;  $\text{ctr} \leftarrow \text{nextCtr}(V_j)$ 
 $v \leftarrow (j, \text{RECV}, \text{ctr}, \text{eid}, v'.\text{label})$ ;  $V_j \leftarrow V_j \cup \{v\}$ 
 $E \leftarrow E \cup \{(v', v)\}$ ; Return  $(n, V_1, \dots, V_n, E)$ 

```

Fig. 3: Functions to initialize a causality graph and add send and receive events.

where $\Pr[\text{bad}^{\mathcal{G}}]$ denotes the probability of setting the flag `bad` in game $\mathcal{G}$.

3 Causality graphs

A causality graph is a data structure that unambiguously identifies all causal information in a group chat. Formally, the causality graph $\mathcal{G} = (n, V_1, V_2, \dots, V_n, E)$ for a group chat among $n \in \mathbb{N}$ users is a directed graph, where the vertex sets $V_1, V_2, \dots, V_n$ are pairwise disjoint and E is the edge set. The set $V = \bigcup_{i=1}^n V_i$ contains all the vertices in the graph; each V_i can be efficiently retrieved from V . For brevity, we use the notations $(n, V_1, V_2, \dots, V_n, E)$ and (n, V, E) interchangeably.

Each vertex in a causality graph represents a sending or receiving action. A vertex v is a tuple $(v.\text{user}, v.\text{type}, v.\text{ctr}, v.\text{eid}, v.\text{label})$, where $v.\text{user}$ identifies the user that performed the action, $v.\text{type} \in \{\text{SEND}, \text{RECV}\}$ indicates whether the action is a send or receive event, $v.\text{ctr}$ is the user's local action counter, $v.\text{eid}$ is a send-event identifier shared by a send vertex and its corresponding receive vertices, and $v.\text{label}$ is the graph label. For every send vertex v , the value $v.\text{eid}$ is unique among the send vertices in $V_{v.\text{user}}$; consequently, $(v.\text{user}, v.\text{eid})$ uniquely identifies a send vertex v . If a receive vertex corresponds to a send vertex in V_i , it carries the same send-event identifier, and $(i, v.\text{eid})$ uniquely identifies the matching send vertex. Since ctr is incremented each time a user performs an action, $(v.\text{user}, v.\text{ctr})$ uniquely identifies a vertex v in a causality graph. An edge represents the delivery of a message. Formally, an edge is an ordered pair of vertices (u, v) indicating a directed edge from u to v where $u \in V_i, v \in V_j$, and $i, j \in [n]$; we permit $i = j$ for a checkpoint creator's later application of its own checkpoint.

Graph functions. We use three functions to construct and update a causality graph. The function `InitGraph` initializes the graph; `AddSendVrtx` and `AddRecvVrtx` add send and receive vertices, respectively. Their pseudocode appears in Fig. 3. The graph-initialization function `InitGraph` takes the size of the group n as input and returns an empty causality graph consisting of empty per-user vertex sets V_i and an empty edge set E .

The `AddSendVrtx` function takes a causality graph $\mathcal{G}$, a sender i , a send-event identifier `eid`, and a graph label `label` as inputs. If V_i already contains a send vertex with send-event identifier `eid`, the function

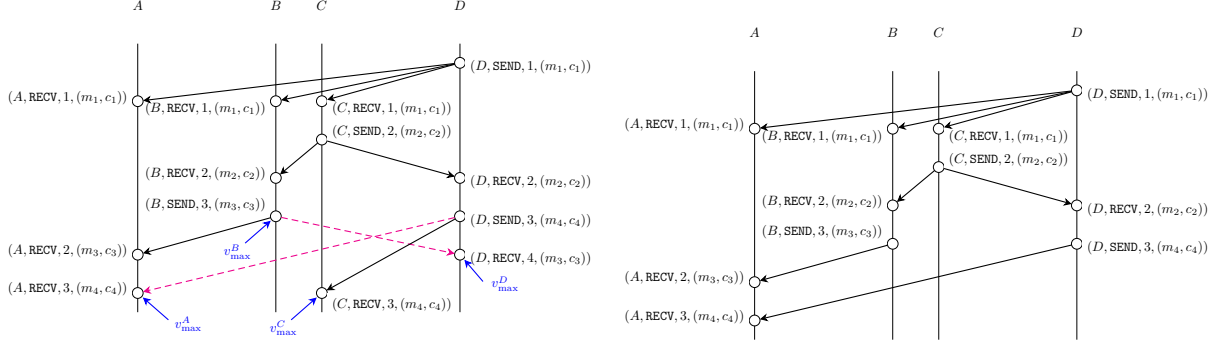


Fig. 4: **Left pane:** A causality graph $\mathcal{G} = (4, V, E)$ for a group chat among four users. The two dashed magenta edges are concurrent deliveries. **Right pane:** The subgraph $\mathcal{G}_A^{\leq}$ corresponding to the graph $\mathcal{G}$ on the left.

returns $\mathcal{G}$ unchanged. Otherwise, it computes user i 's next local action counter, one greater than the largest counter in V_i or 1 if V_i is empty, uses it to create a new send vertex, and adds the vertex to V_i .

The `AddRecvVrtx` function takes a causality graph $\mathcal{G}$, a sender i , a receiver j , and the send-event identifier `eid` as inputs. It looks for the unique send vertex in V_i with send-event identifier `eid`. If no such vertex exists or the identifier is ambiguous, the function returns $\mathcal{G}$ unchanged. Otherwise, it computes user j 's next local action counter, uses it in a new receive vertex, and inserts the vertex together with the delivery edge from the matching send vertex. In graph diagrams, we omit the send-event identifier `v.eid` and display vertices as $(v.\text{user}, v.\text{type}, v.\text{ctr}, v.\text{label})$.

Graph validity. We say a causality graph $\mathcal{G}$ is valid if there exists a sequence of calls to `AddSendVrtx` and `AddRecvVrtx` that creates it from an empty graph. Equivalently, every V_i contains only user- i send and receive vertices with consecutive counters, and its send vertices have unique identifiers. Every receive has exactly one incoming edge from a send with the same identifier and label, and there are no other edges. Finally, no cycle can be formed by following users' timelines in increasing counter order and traversing delivery edges. We provide an example of a valid causality graph for a group chat among four users in Fig. 4.

Ordering over events. A valid causality graph $\mathcal{G} \leftarrow (n, V_1, \dots, V_n, E)$ imposes on each set V_i a strict total order $<_\ell$ reflecting user i 's local order: for all $v_0, v_1 \in V_i$, $v_0 <_\ell v_1 \iff v_0.\text{ctr} < v_1.\text{ctr}$. The graph $\mathcal{G}$ also imposes on V the strict partial order $<$ defined as the transitive closure of all local orders and the edge-induced relations (i.e., for each $(u, v) \in E$ we have $u < v$; and for $u_0, u_1 \in V_i$ and $v \in V_j$, if $u_0 <_\ell u_1$ and $(u_1, v) \in E$ then $u_0 < v$). The partial order $<$ is the same as Lamport's "happened before" relation [Lam78]. We use $\leq$ to denote the reflexive closure of $<$, i.e., $u \leq v \iff u < v$ or $u = v$.

For a causality graph $\mathcal{G} \leftarrow (n, V, E)$ and $j \in [n]$, we define $v_{\max}^j(\mathcal{G}) \leftarrow \max(V_j, <_\ell)$ to be the largest element of V_j with respect to $<_\ell$, i.e., the latest action performed by user j . We define $\mathcal{G}_j^{\leq} \leftarrow (n, V_j^{\leq}, E_j^{\leq})$ where $V_j^{\leq} \leftarrow \{v \in V : v \leq v_{\max}^j\}$ and $E_j^{\leq} \leftarrow \{(u, v) \in E : u, v \in V_j^{\leq}\}$. The graph $\mathcal{G}_j^{\leq}$ is the subgraph of $\mathcal{G}$ that contains j 's latest action as well as all events that happened before the latest action performed by j . If $V_j = \emptyset$, then $v_{\max}^j = \perp$ and $\mathcal{G}_j^{\leq}$ is empty. We illustrate $v_{\max}^j$ and $\mathcal{G}_A^{\leq}$ in a four-user group chat in Fig. 4.

Concurrent edges. Two edges $(u_1, v_1), (u_2, v_2) \in E$ are concurrent either if they are outgoing edges from the same vertex or (1) they originate from different senders, i.e., $u_1 \in V_i$ and $u_2 \in V_j$ where $i \neq j$, and (2) the vertices u_1, v_1, u_2, v_2 are not totally ordered, i.e., $v_1 \not\prec u_2$ and $v_2 \not\prec u_1$.

Graph membership and subgraphs. Let $\mathcal{G} = (n, V_1, \dots, V_n, E)$ be a causality graph and $V \leftarrow \bigcup_{i \in [n]} V_i$. The elements of $\mathcal{G}$ are the vertices and the edges of $\mathcal{G}$, i.e., $V \cup E$, and we write $x \in \mathcal{G}$ for $x \in V \cup E$. Thus $x \in \mathcal{G}$ means either that x is a vertex, in which case $x \in V$, or that x is an edge, in which case $x \in E$. We write $\mathcal{G}_0 \subseteq \mathcal{G}_1$ if every element of $\mathcal{G}_0$ is an element of $\mathcal{G}_1$, i.e., if $V_0 \subseteq V_1$ and $E_0 \subseteq E_1$.

$$\begin{aligned}
(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) &\leftarrow^s \text{GC.Init}(n) \\
(st_i, \mathcal{G}_i^l, c) &\leftarrow^s \text{GC.Send}(i, st_i, \mathcal{G}_i^l, m) \\
(st_j, \mathcal{G}_j^l, m, \text{ctr}) &\leftarrow \text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)
\end{aligned}$$

Fig. 5: Syntax of a group chat.

Graph union and subtraction. Given graphs $\mathcal{G}_0 = (n, V_0, E_0)$ and $\mathcal{G}_1 = (n, V_1, E_1)$, we define $\mathcal{G}_0 \cup \mathcal{G}_1$ as $(n, V_0 \cup V_1, E_0 \cup E_1)$. Similarly, we define $\mathcal{G}_0 - \mathcal{G}_1$ as $(n, V_0 \setminus V_1, E_0 \setminus E_1)$.

4 Group chats

A static group chat allows a group of n users to communicate with each other. Each user $i \in [n]$ can send messages to the group and receive messages sent by any other user $j \in [n] \setminus \{i\}$. In an encrypted group chat, a sender encrypts each message before sending it to the group, and each receiver decrypts the corresponding ciphertext. Each user maintains local protocol state and a local causality graph.

Syntax. We augment the standard bidirectional-channel syntax with *local causality graphs* to capture causal dependencies. The local graph $\mathcal{G}_i^l$ represents the causal information derived by a user i based on their local protocol execution. A local graph is itself a causality graph, so the functions of Fig. 3 apply to it, provided that every edge in it has both of its endpoints in the graph and that the vertex set of the user performing the action contains all of that user’s events.

We give syntax for a group chat in Fig. 5. A group chat GC specifies algorithms GC.Init, GC.Send, and GC.Recv where GC.Recv is deterministic. Associated with GC is a message space GC.MS. The initialization algorithm GC.Init takes the group size $n \in \mathbb{N}$, with $n \geq 1$, as input and returns, for each user $i \in [n]$, an initial state $st_i \in \{0, 1\}^*$ and a local graph $\mathcal{G}_i^l \in \{0, 1\}^* \cup \{\perp\}$. The send algorithm GC.Send takes the sender’s identifier $i \in [n]$, state st_i , local graph $\mathcal{G}_i^l$, and a message $m \in \text{GC.MS}$, and returns the updated state st_i , the updated local graph $\mathcal{G}_i^l$, and a ciphertext c . Finally, the receive algorithm GC.Recv takes the receiver’s identifier j , state st_j , and local graph $\mathcal{G}_j^l$, together with the sender’s identifier i and a ciphertext c . It returns the receiver’s updated state and local graph, a message $m \in \text{GC.MS} \cup \{\perp\}$, and a corresponding value $\text{ctr} \in \mathbb{N} \cup \{\perp\}$ of the sender’s send counter; $m = \perp$ denotes rejection. When an accepted receive is recorded in a causality graph, this counter value serves as the send-event identifier of the corresponding send vertex. We use the standard rejection convention that failed receives are state-preserving: if GC.Recv returns $m = \perp$, then the returned state and local graph equal the input state and local graph. Ordinary bidirectional channels are the special case in which every $\mathcal{G}_i^l$ is $\perp$.

4.1 Correctness

We define two correctness notions for group chats: message correctness and graph correctness. Message correctness requires that all honestly sent messages can be successfully received with the correct sending index. Graph correctness requires that when a user receives a message, their local view of the conversation correctly captures the causal context of that message, i.e., they know which messages causally preceded it. In the strongest sense, graph correctness requires that each user’s local view includes all events that happened before the latest action performed by them. We also capture weaker variants.

We formally define the message correctness of a group chat GC consisting of n users using the game $\mathcal{G}^{\text{CORR-}m}$ shown in Fig. 6. The advantage of $\mathcal{A}_{\text{CORR-}m}$ against the correctness of GC is given by $\text{Adv}_{\text{GC}, n}^{\text{CORR-}m}(\mathcal{A}_{\text{CORR-}m}) = \Pr[\mathcal{G}_{\text{GC}, n}^{\text{CORR-}m}(\mathcal{A}_{\text{CORR-}m})]$. Adversary $\mathcal{A}_{\text{CORR-}m}$ is given the initial states and local graphs of all users, the SEND and RECV oracles, and is tasked with setting the win flag.

The SEND oracle takes a user i and a message m as inputs and returns the updated state st_i , local graph $\mathcal{G}_i^l$, and ciphertext c obtained by running $\text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$. It uses the counter ctr_i to track the index of messages sent by the user i , incrementing it each time the user sends a message. The oracle records the input and output of each query by storing m in $M[i, \text{ctr}_i]$ and c in $C[i, j, \text{ctr}_i]$ for each $j \in [n] \setminus \{i\}$. All of these C entries contain the same broadcast ciphertext and can be consumed independently.

Games $\mathcal{G}_{\text{GC},n}^{\text{CORR-}m}(\mathcal{A}_{\text{CORR-}m})$ and $\mathcal{G}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}})$	
win $\leftarrow$ false ; $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC.Init}(n)$	
ctr _j $\leftarrow$ 0 $\forall j \in [n]$	
$\mathcal{A}_{\text{CORR-}m}^{\text{SEND,RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	// CORR- m
$\mathcal{G} \leftarrow \text{InitGraph}(n)$; $\mathcal{A}_{\text{CORR-}\mathcal{G}}^{\text{SEND,RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	// CORR- $\mathcal{G}$
Return win	
<u>SEND(i, m)</u> // $i \in [n], m \in \text{GC.MS}$	
ctr _i $\leftarrow$ ctr _i + 1 ; $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$	
$M[i, \text{ctr}_i] \leftarrow m$	// CORR- m
$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$	// CORR- $\mathcal{G}$
$C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$	
Return $(st_i, \mathcal{G}_i^l, c)$	
<u>RECV(i, j, ctr)</u> // $i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$	
require $C[i, j, \text{ctr}] \neq \perp$	
$c \leftarrow C[i, j, \text{ctr}]$	
$(st_j, \mathcal{G}_j^l, m, \text{ctr}') \leftarrow \text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$	
If $(\text{ctr}', m) \neq (\text{ctr}, M[i, \text{ctr}])$ then win $\leftarrow$ true	// CORR- m
If $m = \perp$ then return $\perp$	// CORR- $\mathcal{G}$
$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$	// CORR- $\mathcal{G}$
If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then	// CORR- $\mathcal{G}$
win $\leftarrow$ true	// CORR- $\mathcal{G}$
$C[i, j, \text{ctr}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, m, \text{ctr}')$	

Fig. 6: Games $\mathcal{G}^{\text{CORR-}m}$ and $\mathcal{G}^{\text{CORR-}\mathcal{G}}$ defining message and graph correctness of GC. Both games use parameter n ; graph correctness additionally uses parameters **restrict** and **redact**. Unannotated lines are shared; right-hand comments mark game-specific lines.

The RECV oracle takes (i, j, ctr) as input and requires that $C[i, j, \text{ctr}] \neq \perp$. This means that the oracle SEND recorded this ciphertext for delivery from i to j and that the corresponding delivery record has not been consumed. The oracle runs $\text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, C[i, j, \text{ctr}])$ to obtain $st_j, \mathcal{G}_j^l, m, \text{ctr}'$. The input ctr identifies the selected honest send, whereas ctr' is returned by the algorithm. If the received message or index is different from what was sent, the oracle sets the win flag. Finally, the oracle consumes the selected table entry and returns $\mathcal{G}_j^l, m, \text{ctr}'$.

We define the graph correctness of a group chat GC consisting of n users with respect to the restricted graph function **restrict** and the redaction function **redact** using the game $\mathcal{G}^{\text{CORR-}\mathcal{G}}$ shown in Fig. 6. The advantage of $\mathcal{A}_{\text{CORR-}\mathcal{G}}$ against the correctness of GC is given by $\text{Adv}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}}) = \Pr[\mathcal{G}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}})]$. Adversary $\mathcal{A}_{\text{CORR-}\mathcal{G}}$ is given the initial states and local graphs of all users, the SEND and RECV oracles, and must set the win flag.

The SEND oracle is the same as in $\mathcal{G}^{\text{CORR-}m}$ except that instead of storing the input message m in the table M , the oracle updates the “global” graph $\mathcal{G}$, initialized by $\text{InitGraph}(n)$, by adding the current send event using the function $\text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ defined in Fig. 3. The global graph $\mathcal{G}$ is the game-maintained causality graph that records all causal relationships in the conversation and serves as the ground truth against which each user’s local graph $\mathcal{G}_i^l$ is compared in the graph correctness game. The RECV oracle invokes GC.Recv on the selected recorded ciphertext. If GC.Recv rejects, the oracle returns $\perp$ without consuming the delivery entry. After acceptance, the oracle adds the corresponding receive vertex to $\mathcal{G}$. Instead of checking the returned message and index, the oracle sets win if $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$. We define the syntax and example variants of these functions below.

Restricted graph functions. We parameterize $\mathcal{G}^{\text{CORR-}\mathcal{G}}$ with a restricted graph function **restrict**. Given a causality graph $\mathcal{G}$ and a user j , the function returns the part of $\mathcal{G}$ that j must know. In particular, $\text{restrict}(\mathcal{G}, j)$ is a subgraph of $\mathcal{G}_j^{\leq}$: it contains selected vertices at or before j ’s latest action and the edges

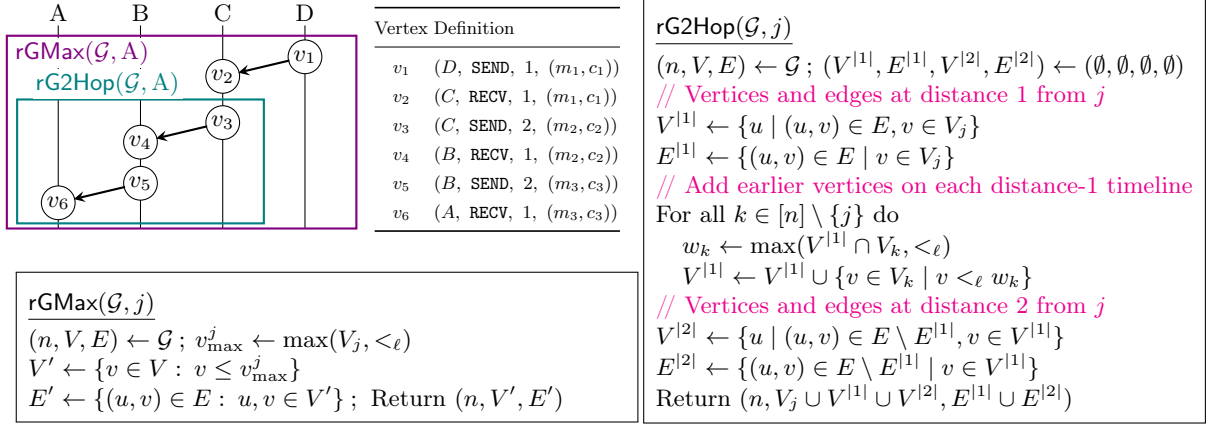


Fig. 7: **Left pane:** Restrictions $\text{rGMax}(\mathcal{G}, A)$ (violet) and $\text{rG2Hop}(\mathcal{G}, A)$ (teal) of the displayed graph, with vertex definitions (top); function rGMax (bottom). **Right pane:** Function rG2Hop .

between those vertices. The graph-correctness game compares the receiver's (redacted) local graph against the restricted (redacted) global graph $\text{restrict}(\text{redact}(\mathcal{G}), j)$ and sets win if they differ.

In Fig. 7 we define two sample restricted graph functions and provide corresponding visual representations. The function rGMax captures the full-history target: $\text{rGMax}(\mathcal{G}, j)$ returns $\mathcal{G}_j^{\leq}$, which contains every event at or before j 's latest action and the edges between those events. The function rG2Hop captures a narrower target. For each message that j has received from i , it contains i 's timeline through the corresponding send and the immediate incoming dependencies of that timeline. This target supplies the causal information underlying the UI feature in Fig. 2, where long-pressing a message surfaces the sender's view.

Concretely, $\text{rG2Hop}(\mathcal{G}, j)$ includes the following. First, it includes j 's own vertices V_j (this is the distance 0 vertex set $V^{[0]}$). Second, for each message j has received from some party i , it includes the corresponding edge (to the set $E^{[1]}$) as well as all the vertices in V_i up to that send (these vertices form the distance 1 vertex set $V^{[1]}$). Finally, it collects all sending vertices that are connected to receiving vertices in $V^{[1]}$ and the corresponding edges (forming the distance 2 sets $V^{[2]}$ and $E^{[2]}$). Jointly, the graph $(n, V_j \cup V^{[1]} \cup V^{[2]}, E^{[1]} \cup E^{[2]})$ provides j all the information it needs to reconstruct the local view of any party who has communicated directly with j , without requiring knowledge of unrelated parts of the global conversation.

Redaction functions. While restrict controls which vertices and edges of the (redacted) global graph a receiver must know, redact controls how much information from each vertex label the receiver must know. The game applies redact before restrict selects the required subgraph. For example, consider Fig. 7. Under rGMax and the identity redaction function ($\text{redact} = \mathcal{I}$), A must know not only that B and C received some messages but also the contents of those messages, even though A has not directly received them. This may be too strong, so we use weaker redaction functions to remove such information from both the global graph $\mathcal{G}$ and the reconstructed local graph $\mathcal{G}_A^l$, so that A is only required to know the redacted information.

We consider two kinds of redaction functions, global and selective. A global redaction function takes a causality graph $\mathcal{G}$ and redacts every vertex label. A selective redaction function also takes a user i as input and redacts vertex labels selectively with respect to V_i ; in a game, this argument is the current receiver and is suppressed from the notation. A redaction preserves graph structure while possibly changing vertex-label fields, returning a *redacted causality graph*.

We consider two redaction functions, one global and one selective. The selective redaction function redactLblS takes a causality graph $\mathcal{G}$ and a user i and removes the `label` field from vertices $v \in V \setminus V_i$. The global redaction function redactLbl takes a causality graph $\mathcal{G}$ and removes the `label` field from all vertices $v \in V$. These functions act pointwise on vertices and edges; we use the same notation for their natural extensions to causality graphs. Their pseudocode is given in Fig. 8.

```

redactLbIS( $\mathcal{G}, i$ ) //  $i \in [n]$ 
 $(n, V, E) \leftarrow \mathcal{G}$ 
Let  $R_i(v) = v$  if  $v \in V_i$ , and  $R_i(v) = \text{redactLbIVrtx}(v)$  otherwise
 $V^{\text{redacted}} \leftarrow \{R_i(v) \mid v \in V\}$ 
 $E^{\text{redacted}} \leftarrow \{(R_i(u), R_i(v)) \mid (u, v) \in E\}$ 
Return  $(n, V^{\text{redacted}}, E^{\text{redacted}})$ 

redactLbI( $\mathcal{G}$ )
 $(n, V, E) \leftarrow \mathcal{G}$ 
 $V^{\text{redacted}} \leftarrow \{\text{redactLbIVrtx}(v) \mid v \in V\}$ 
 $E^{\text{redacted}} \leftarrow \{(\text{redactLbIVrtx}(u), \text{redactLbIVrtx}(v)) \mid (u, v) \in E\}$ 
Return  $(n, V^{\text{redacted}}, E^{\text{redacted}})$ 



---


redactLbIVrtx( $v$ ) Return  $(v.\text{user}, v.\text{type}, v.\text{ctr}, v.\text{eid}, \perp)$ 

```

Fig. 8: Redaction functions `redactLbI` and `redactLbIS`, and helper function `redactLbIVrtx`.

<u>$\text{GC}_{\text{naive-max}}.\text{Init}(n)$</u> $st_i \leftarrow 0, \mathcal{G}_i^l \leftarrow \text{InitGraph}(n) \forall i \in [n]$ Return $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l)$ <u>$\text{GC}_{\text{naive-max}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$</u> $st_i \leftarrow st_i + 1$ $\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$ $\text{ctr} \leftarrow st_i; \mathcal{G}^\Delta \leftarrow \mathcal{G}_i^l$ $p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr})$ Return $(st_i, \mathcal{G}_i^l, p)$ <u>$\text{GC}_{\text{naive-max}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, p)$</u> $(\mathcal{G}^\Delta, m, \text{ctr}) \leftarrow p$ $\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$ $\mathcal{G}_j^l \leftarrow \text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr})$ Return $(st_j, \mathcal{G}_j^l, m, \text{ctr})$	<u>$\text{GC}_{\text{max}}.\text{Init}(n)$</u> <u>$\text{GC}_{\text{max}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, p)$</u> // Both algorithms are // as in $\text{GC}_{\text{naive-max}}$. <u>$\text{GC}_{\text{max}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$</u> $st_i \leftarrow st_i + 1$ $\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$ For all $j \in [n] \setminus \{i\}$ do $\mathcal{G}_j \leftarrow \text{rGMax}(\mathcal{G}_i^l, j)$ $\mathcal{G}^\Delta \leftarrow \bigcup_{j \in [n] \setminus \{i\}} (\mathcal{G}_i^l - \mathcal{G}_j)$ $\text{ctr} \leftarrow st_i; p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr})$ Return $(st_i, \mathcal{G}_i^l, p)$	<u>$\text{GC}_{2\text{Hop}}.\text{Init}(n)$</u> <u>$\text{GC}_{2\text{Hop}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, p)$</u> // Both algorithms are as in $\text{GC}_{\text{naive-max}}$. <u>$\text{GC}_{2\text{Hop}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$</u> $st_i \leftarrow st_i + 1$ $\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$ $(n, V, E) \leftarrow \mathcal{G}_i^l$ $\mathcal{G}_i \leftarrow (n, V_i, \{\})$ $\cup (n, \{\}, \{(x, y) \in E : y \in V_i\})$ $\cup (n, \{x : (x, y) \in E \wedge y \in V_i\}, \{\})$ For all $j \in [n] \setminus \{i\}$ do $\mathcal{G}_j \leftarrow \text{rG2Hop}(\mathcal{G}_i^l, j)$ $\mathcal{G}^\Delta \leftarrow \bigcup_{j \in [n] \setminus \{i\}} (\mathcal{G}_i - \mathcal{G}_j)$ $\text{ctr} \leftarrow st_i; p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr})$ Return $(st_i, \mathcal{G}_i^l, p)$
--	--	--

Fig. 9: **Left pane:** Group chat $\text{GC}_{\text{naive-max}}$, graph-correct for rGMax . **Center pane:** Group chat GC_{max} , graph-correct for rGMax . **Right pane:** Group chat $\text{GC}_{2\text{Hop}}$, graph-correct for rG2Hop .

Together, different choices of `restrict` and `redact` give different graph-correctness definitions. For each fixed `restrict` $\in \{\text{rGMax}, \text{rG2Hop}\}$, retaining more label fields yields a stronger notion, as summarized in Fig. 12.

5 Unencrypted group chats

In this section, we build graph-correct unencrypted channels. Focusing on unencrypted channels lets us identify, independent of cryptography, the properties of the causal information each broadcast message must make available for receivers to reconstruct the appropriate portion of the global transcript. In particular, we fix `redact` = `redactLbI` (retaining only the causal skeleton)³ and determine the graph evidence required to achieve graph correctness for each choice of restricted graph function (rGMax , rG2Hop).

³ The other redaction choices require carrying the appropriate messages or ciphertexts in local graph labels. We focus on `redactLbI` here to isolate the causal skeleton from label-encoding choices.

5.1 Graph correctness with respect to rGMax

We begin by building group chats that are graph-correct with respect to the restricted graph function rGMax. As a warm-up, we first present a naive construction that achieves graph correctness but incurs large communication overhead. We then modify the construction to minimize communication overhead and prove optimality.

Naive construction $\text{GC}_{\text{naive-max}}$. We present our naive solution in Fig. 9. The idea is to have each user record all send and receive events in their local graph, and attach their entire local graph to each outgoing message. The receiver simply performs a union of the received graph with their own local graph.

The initialization algorithm $\text{GC}_{\text{naive-max}}.\text{Init}(n)$ sets each user's state to 0 and local graph to the empty graph. The send algorithm $\text{GC}_{\text{naive-max}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ increments the sender's state, and adds the current sending event to the sender's local graph $\mathcal{G}_i^l$ by running $\text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$. It returns st_i , $\mathcal{G}_i^l$, and the payload p containing $\mathcal{G}_i^l$, m , and st_i .

The receive algorithm $\text{GC}_{\text{naive-max}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, p)$ parses p into $\mathcal{G}^\Delta$, m , ctr , and merges $\mathcal{G}^\Delta$ into the receiver's local graph $\mathcal{G}_j^l$, then adds the current receive event to $\mathcal{G}_j^l$ by running $\text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr})$. Finally, it returns st_j , the updated local graph $\mathcal{G}_j^l$, m , and ctr .

Theorem 1. Let $\text{GC}_{\text{naive-max}}$ be the group chat scheme shown in Fig. 9 and $n \in \mathbb{N}$. Let rGMax be the restricted graph function defined in Fig. 7 and redactLbl be the redaction function defined in Section 4.1. Let $\mathcal{A}_{\text{CORR-G}}$ be an adversary against the CORR-G security of $\text{GC}_{\text{naive-max}}$ with respect to n , rGMax, and redactLbl . Then,

$$\text{Adv}_{\text{GC}_{\text{naive-max}}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-G}}(\mathcal{A}_{\text{CORR-G}}) = 0.$$

Proof sketch. We show that the invariant $\text{redactLbl}(\mathcal{G}_i^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), i)$ holds for all $i \in [n]$ throughout the game. At initialization all graphs are empty, so the invariant holds trivially. Each $\text{SEND}(i, m)$ call adds corresponding sending vertices to $\mathcal{G}_i^l$ and $\mathcal{G}$ that coincide after applying redactLbl , preserving the invariant. Each $\text{RECV}(i, j, \text{ctr})$ call adds a new receiving vertex v_{RECV} and the edge $(v_{\text{SEND}}, v_{\text{RECV}})$ connecting v_{RECV} to the corresponding sending vertex v_{SEND} in $\mathcal{G}$. This extends $\text{rGMax}(\text{redactLbl}(\mathcal{G}), j)$ by (i) the receive event itself, and (ii) all causal predecessors of v_{SEND} (i.e., all vertices $w \in \mathcal{G} : w \leq v_{\text{SEND}}$). Since the sender includes its entire local graph in $\mathcal{G}^\Delta$, it includes all the predecessors of v_{SEND} , and $\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$ covers (ii). The explicit $\text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr})$ covers (i). Therefore, the invariant is also preserved after each RECV call. The formal proof appears in Appendix A.1.

Including one's entire local graph with every outgoing message is clearly expensive and impractical. We now define the minimum broadcast graph evidence required to satisfy our correctness definitions.⁴

Definition 1 (Send instance). Consider a SEND query in an execution of the graph correctness game $\text{GC}_{\text{CORR-G}, n, \cdot, \cdot}$ for a group chat GC among $n > 1$ users. The send instance defined by the SEND query is the tuple $\Sigma = (i, m, \mathcal{G}_i^l, \text{ctr}_i, \mathcal{G}^\Delta)$ consisting of the inputs of the query, the sender's local graph $\mathcal{G}_i^l$ and game-maintained send counter ctr_i immediately after the query, and the causality graph $\mathcal{G}^\Delta$ contained in the returned payload.

Definition 2 (Consistent state). A tuple $(\mathcal{G}, \mathcal{G}_j^l)$ consisting of a global graph and a local graph is consistent with a send instance $\Sigma = (i, m, \mathcal{G}_i^l, \text{ctr}_i, \mathcal{G}^\Delta)$ if some execution of the graph correctness game makes a SEND query whose send instance is Σ and immediately after which the global graph is $\mathcal{G}$ and user $j \neq i$ has local graph $\mathcal{G}_j^l$.

Definition 3 (Necessary element). Let GC be a group chat among $n > 1$ users and $i \in [n]$ be a sender. Let $\Sigma = (i, m, \mathcal{G}_i^l, \text{ctr}_i, \mathcal{G}^\Delta)$ be a send instance in the graph correctness game of GC with respect to a restricted graph function restrict and a redaction function redact . An element $x \in \mathcal{G}^\Delta$ is necessary if there exist a receiver $j \in [n] \setminus \{i\}$ and global graph $\mathcal{G}$ such that $(\mathcal{G}, \mathcal{G}_j^l)$ is consistent with Σ , $x \notin \text{redact}(\mathcal{G}_j^l)$, and $x \in \text{restrict}(\text{redact}(\mathcal{G}_{i \rightarrow j}), j)$ where $\mathcal{G}_{i \rightarrow j} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr}_i)$.

⁴ We assume users transmit their causality information by including a graph $\mathcal{G}^\Delta$ in each send payload; our definitions are with respect to $\mathcal{G}^\Delta$.

Definition 4 (Optimal graph). The graph $\mathcal{G}^\Delta$ of a send instance $\Sigma = (i, m, \mathcal{G}_i^l, \text{ctr}_i, \mathcal{G}^\Delta)$ is optimal with respect to restrict and redact if every element $x \in \mathcal{G}^\Delta$ is necessary.

Definition 5 (Graph-optimal group chat). A group chat GC is graph-optimal with respect to restrict and redact if for every $n > 1$, the graph $\mathcal{G}^\Delta$ of every send instance of GC in $\mathcal{G}_{\text{GC}, n, \text{restrict}, \text{redact}}^{\text{CORR-G}}$ is optimal.

Optimal construction GC_{max} . We now present our construction GC_{max} that achieves graph correctness with respect to rGMax and is graph-optimal. The idea is to have each party's sends implicitly acknowledge the events in their local graph. When party j receives a payload from i , j learns which events i is aware of and updates its record of i 's state accordingly. Then, on its next send, j only needs to include the portions of its local graph not yet known to all other members. This idea was also used to optimize Chen and Fischlin's generic causal channel compiler [CF24].

The algorithms $\text{GC}_{\text{max}}.\text{Init}$ and $\text{GC}_{\text{max}}.\text{Recv}$ are defined identically to $\text{GC}_{\text{naive-max}}.\text{Init}$ and $\text{GC}_{\text{naive-max}}.\text{Recv}$. The algorithm $\text{GC}_{\text{max}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ first increments st_i and adds the current send event to $\mathcal{G}_i^l$. Then it computes the graph $\mathcal{G}^\Delta$ to be included in the payload as follows. For each $j \in [n] \setminus \{i\}$, it derives the graph $\mathcal{G}_j \leftarrow \text{rGMax}(\mathcal{G}_i^l, j)$, i.e., it computes the restricted graph function with respect to its own local graph. The graph $\mathcal{G}_j$ is the portion of $\mathcal{G}_i^l$ that i knows j already has. This means that $\mathcal{G}_i^l - \mathcal{G}_j$ contains the portion of $\mathcal{G}_i^l$ that j has not yet confirmed receipt of. It then sets $\mathcal{G}^\Delta$ to be the union of $\mathcal{G}_i^l - \mathcal{G}_j$ for all $j \in [n] \setminus \{i\}$, and returns st_i , $\mathcal{G}_i^l$, and the payload $p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr})$.

Theorem 2. Let GC_{max} be the group chat scheme shown in Fig. 9 and $n > 1$. Let rGMax be the restricted graph function defined in Fig. 7 and redactLbl be the redaction function defined in Section 4.1. Let $\mathcal{A}_{\text{CORR-G}}$ be an adversary against the CORR-G security of GC_{max} with respect to n , rGMax, and redactLbl. Then,

$$\text{Adv}_{\text{GC}_{\text{max}}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-G}}(\mathcal{A}_{\text{CORR-G}}) = 0.$$

Proof sketch. We show the invariant $\text{redactLbl}(\mathcal{G}_i^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), i)$ holds for all $i \in [n]$ throughout the game. Initialization and each SEND are handled identically to the proof of Theorem 1. For RECV(i, j, ctr), since $\text{GC}_{\text{max}}.\text{Recv}$ is identical to $\text{GC}_{\text{naive-max}}.\text{Recv}$, it suffices to show $\text{redactLbl}(\mathcal{G}_j^l \cup \mathcal{G}^\Delta) = \text{redactLbl}(\mathcal{G}_j^l \cup \mathcal{G}_i^l)$; the invariant follows by the same argument. Since for every receiver $j \in [n] \setminus \{i\}$, i 's estimate $\mathcal{G}_j$ of j 's local graph satisfies $\text{redactLbl}(\mathcal{G}_j) \subseteq \text{redactLbl}(\mathcal{G}_i^l)$, every element of $\text{redactLbl}(\mathcal{G}_i^l)$ that is missing from $\text{redactLbl}(\mathcal{G}_j^l)$ appears in $\text{redactLbl}(\mathcal{G}^\Delta)$. The formal proof appears in Appendix A.2.

Theorem 3. Let GC_{max} be the group chat protocol defined in Fig. 9 and let $n > 1$. Then GC_{max} is graph-optimal with respect to rGMax and redactLbl.

Proof. Fix an arbitrary send instance $\Sigma = (i, m, \mathcal{G}_i^l, \text{ctr}_i, \mathcal{G}^\Delta)$ of GC_{max} in $\mathcal{G}_{\text{GC}_{\text{max}}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-G}}$. It suffices to show that $\forall x \in \mathcal{G}^\Delta$, there exist a receiver j and a global graph $\mathcal{G}$ such that $(\mathcal{G}, \mathcal{G}_j^l)$ is consistent with Σ , and (i) $x \notin \text{redactLbl}(\mathcal{G}_j^l)$ and (ii) $x \in \text{rGMax}(\text{redactLbl}(\mathcal{G}_{i \rightarrow j}), j)$.

Consider some $x \in \mathcal{G}^\Delta$. By construction of $\mathcal{G}^\Delta$, there exists a $j \in [n] \setminus \{i\}$ such that $x \in \mathcal{G}_i^l$ and $x \notin \mathcal{G}_j$. Since the interim graph $\mathcal{G}_j$ computed by $\text{GC}_{\text{max}}.\text{Send}$ satisfies $\mathcal{G}_j = \text{rGMax}(\mathcal{G}_i^l, j)$, we have

$$x \notin \text{rGMax}(\mathcal{G}_i^l, j). \quad (1)$$

Consider an execution of the graph correctness game for GC_{max} with respect to rGMax and redactLbl such that the sequence of SEND and RECV queries corresponds exactly to the events of $\mathcal{G}_i^l$ ordered by $<$. Then the global graph $\mathcal{G}$ of that execution satisfies $\text{redactLbl}(\mathcal{G}) = \mathcal{G}_i^l$. Let $\mathcal{G}_j^l$ be j 's local graph immediately after the SEND query. Then $(\mathcal{G}, \mathcal{G}_j^l)$ is consistent with Σ . Moreover, by the invariant established in the proof of Theorem 2,

$$\text{redactLbl}(\mathcal{G}_j^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), j). \quad (2)$$

Using $\text{redactLbl}(\mathcal{G}) = \mathcal{G}_i^l$ in Equation (2) gives

$$\text{redactLbl}(\mathcal{G}_j^l) = \text{rGMax}(\mathcal{G}_i^l, j). \quad (3)$$

Combining Equations (1) and (3) establishes (i).

Let u be i 's latest send vertex in $\mathcal{G}_i^l (= \text{redactLbl}(\mathcal{G}))$. When x is a vertex, x satisfies $x \leq u$ since $x \in \mathcal{G}_i^l$ and $\mathcal{G}_i^l = \text{rGMax}(\text{redactLbl}(\mathcal{G}), i)$. The graph $\mathcal{G}_{i \rightarrow j}$ is obtained by adding a receiving vertex v to V_j and the edge (u, v) to $\mathcal{G}$. Therefore, $v = v_{\max}^j(\mathcal{G}_{i \rightarrow j})$ and $u < v$. Since $x \leq u$ and $u < v$, we have $x < v$, i.e., $x \in \text{rGMax}(\text{redactLbl}(\mathcal{G}_{i \rightarrow j}), j)$ which establishes (ii). When x is an edge of the form (u', v') , the vertices u' and v' satisfy $u' \leq u$ and $v' \leq u$. Then by the same argument as above, we get $x \in \text{rGMax}(\text{redactLbl}(\mathcal{G}_{i \rightarrow j}), j)$. Therefore x is necessary.

As $x \in \mathcal{G}^\Delta$ was arbitrary, $\mathcal{G}^\Delta$ is optimal, and as Σ was arbitrary, $\text{GC}_{\max}$ is graph-optimal with respect to rGMax and redactLbl .

5.2 Graph correctness with respect to rG2Hop

The optimal construction for rGMax improves over $\text{GC}_{\text{naive-max}}$, but is still quite expensive. It requires every receiver to learn all unconfirmed events that happened before the sender's latest action, including events from and between every other party in the group.

We now present a construction that satisfies graph correctness with respect to rG2Hop . Our construction provides the graph evidence needed to realize the UI feature from [Section 1](#). It is not graph-optimal; we discuss why at the end of this section. The idea is to have each user record their own send/receive events. Then on each send, a user attaches the portion of this record not yet confirmed by all other parties.

The algorithms $\text{GC}_{2\text{Hop}}.\text{Init}$ and $\text{GC}_{2\text{Hop}}.\text{Recv}$ are defined identically to $\text{GC}_{\text{naive-max}}.\text{Init}$ and $\text{GC}_{\text{naive-max}}.\text{Recv}$. The algorithm $\text{GC}_{2\text{Hop}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ first increments st_i and adds the current send event to $\mathcal{G}_i^l$. It computes $\mathcal{G}^\Delta$ as follows. It first computes $\mathcal{G}_i$ which consists of i 's vertices V_i , the edges into them, and the sources of those edges. For each $j \in [n] \setminus \{i\}$ it then computes $\mathcal{G}_j$, the part of its local graph that i knows j already has, as $\mathcal{G}_j \leftarrow \text{rG2Hop}(\mathcal{G}_i^l, j)$. It sets $\mathcal{G}^\Delta$ to be the part of $\mathcal{G}_i$ not yet confirmed by every other party by setting $\mathcal{G}^\Delta \leftarrow \bigcup_{j \neq i} (\mathcal{G}_i - \mathcal{G}_j)$. Finally, it returns the payload $p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr})$.

Theorem 4. *Let $\text{GC}_{2\text{Hop}}$ be the group chat scheme shown in [Fig. 9](#) and $n \in \mathbb{N}$. Let rG2Hop be the restricted graph function defined in [Fig. 7](#) and redactLbl be the redaction function defined in [Section 4.1](#). Let $\mathcal{A}_{\text{CORR-}\mathcal{G}}$ be an adversary against the $\text{CORR-}\mathcal{G}$ security of $\text{GC}_{2\text{Hop}}$ with respect to n , rG2Hop , and redactLbl . Then,*

$$\text{Adv}_{\text{GC}_{2\text{Hop}}, n, \text{rG2Hop}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}}) = 0.$$

Proof sketch. We show the invariant $\text{redactLbl}(\mathcal{G}_i^l) = \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), i)$ holds for all $i \in [n]$ throughout the game. Initialization and each SEND are handled identically to the proof of [Theorem 1](#). For $\text{RECV}(i, j, \text{ctr})$, let v be the current receiving vertex and $u^* \in V_i$ be the corresponding sending vertex. Adding the current receive event (i.e., $v, (u^*, v)$) to $\mathcal{G}$ extends $\text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j)$ by (i) the receive event itself, (ii) i 's events after v_j up to u^* , and (iii) the edges into those events and their source vertices, where v_j is the latest send of i that j has received. We argue that the local graph $\mathcal{G}_j^l$ is also extended identically. The graph $\mathcal{G}_i$ contains all elements in (ii) and (iii). An element of $\mathcal{G}_i$ is not included in $\mathcal{G}^\Delta$ only if it exists in $\text{rG2Hop}(\mathcal{G}_i^l, j)$, which by the invariant implies it exists in $\text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j) = \text{redactLbl}(\mathcal{G}_j^l)$. So every element of (ii) and (iii) that j lacks is in $\mathcal{G}^\Delta$; AddRecvVrtx covers (i). The formal proof appears in [Appendix A.3](#).

Optimality of $\text{GC}_{2\text{Hop}}$. Our $\text{GC}_{2\text{Hop}}$ is not graph-optimal. To see this, consider the example in [Figure 10](#). When A sends v_8 , $\text{rG2Hop}(\mathcal{G}_A^l, C) = \{v_4\}$, so $\text{GC}_{2\text{Hop}}.\text{Send}$ includes v_1 in $\mathcal{G}^\Delta$. However, $v_1 \in \mathcal{G}_C^l$, so $\mathcal{G}^\Delta$ is not optimal per [Definition 4](#).

This is because $\text{GC}_{2\text{Hop}}.\text{Send}$'s estimate of what elements receiver j 's local graph already contains ($\text{rG2Hop}(\mathcal{G}_j^l, j)$) does not consider what the sender can infer from the ctr and eid values in $\mathcal{G}_i^l$. In this example, $v_4.\text{ctr} = 3$ and $v_4.\text{eid} = 1$. Therefore A can infer that C's first send was its third action, so C must have received two messages before sending. In $\mathcal{G}_A^l$, the only sends that precede v_4 are v_1 and v_3 , so A can infer that C received v_1 . B also has v_1 (via e_1), so v_1 is not necessary. The $\text{GC}_{2\text{Hop}}.\text{Send}$ algorithm does not consider such inferences; adding them would cost the simplicity of the algorithm.

The problem lies in the fact that eid is redundant information in a causality graph, since $(v.\text{user}, v.\text{ctr})$ already uniquely identifies a vertex and $v.\text{ctr}$ allows ordering of actions performed by a user. Including

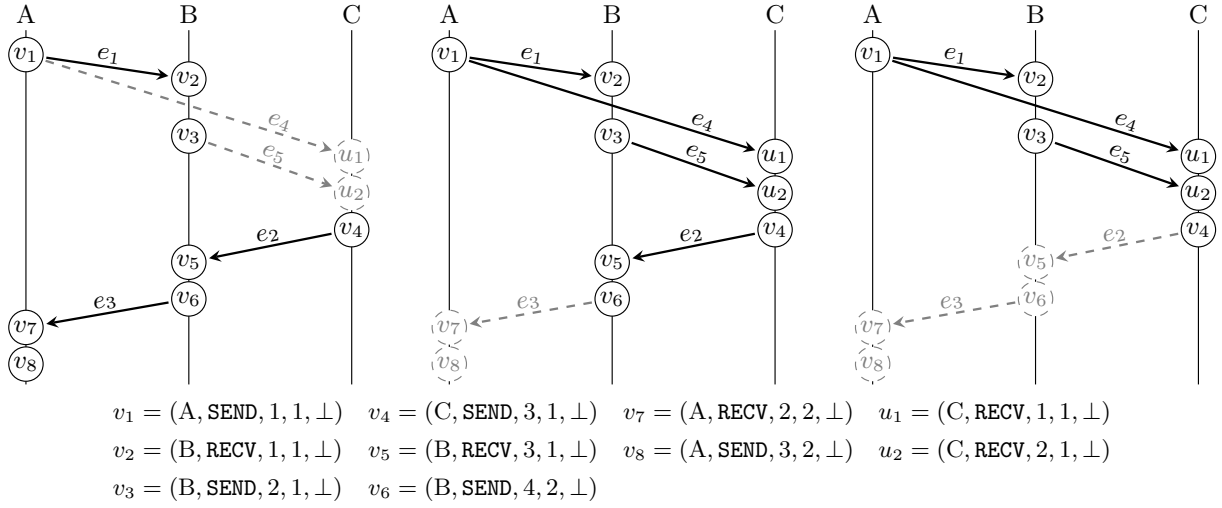


Fig. 10: Counterexample to the graph-optimality of $\text{GC}_{2\text{Hop}}$. Vertices are (user, type, ctr, eid, label). A sends a_1 (v_1); B receives it (v_2) and sends b_1 (v_3); C receives a_1 (u_1) and b_1 (u_2) and sends c_1 (v_4); B receives c_1 (v_5) and sends b_2 (v_6); A receives b_2 (v_7) and sends a_2 (v_8). Solid vertices and edges are in the local graph when A sends a_2 ; dashed ones complete the global graph. **Left pane:** A’s local graph. **Center pane:** B’s local graph. **Right pane:** C’s local graph.

eid in the local graph reveals how many of a user’s earlier actions were sends, and hence reveals the types (send/receive) of events that are not in the sender’s current local graph. Essentially, even though the construction satisfies the graph correctness definition, the sender can learn information beyond its two-hop view.

One way to correct this is to not add eid to local graphs and to update `redactLbl` such that it additionally drops eid. Then $v_4 = (C, \text{SEND}, 3, \perp)$ is also consistent with C’s first two actions being sends that no one has received yet, and the inference above no longer applies. However, now the sender must include the ctr value of its send vertex in the payload, which the receiver uses to identify that vertex. We do not present this construction to maintain consistency between constructions.

6 Encrypted group chats

In this section, we define notions of integrity and causal awareness for encrypted group chats. Our integrity notions adapt standard notions of integrity of ciphertexts and integrity of plaintexts for bidirectional channels and introduce a stronger variant capturing both. Causal awareness requires that an adversary cannot cause a receiver to accept a ciphertext outside its causal context. We show that the causal awareness of a group chat follows from its INT-PCTXT integrity and graph correctness. We apply this result to construct GtE (Graph-then-Encrypt) by composing an unencrypted graph-correct group chat with an encrypted bidirectional channel.

6.1 Integrity of group chats

We define three integrity variants for an n -user group chat GC using game $\mathcal{G}^{\text{INT-}x}$ in the left panel of Fig. 11: plaintext integrity (INT-PTXT), ciphertext integrity (INT-CTXT), and joint plaintext-and-ciphertext integrity (INT-PCTXT). The game is parameterized by $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$. In each group of variant-specific lines, only the line marked with the selected variant is active. The advantage of $\mathcal{A}_{\text{INT-}x}$ against the integrity of GC is given by $\text{Adv}_{\text{GC}, n}^{\text{INT-}x}(\mathcal{A}_{\text{INT-}x}) = \Pr[\mathcal{G}_{\text{GC}, n}^{\text{INT-}x}(\mathcal{A}_{\text{INT-}x})]$. Adversary $\mathcal{A}_{\text{INT-}x}$ is given access to the SEND and RECV oracles and wins by setting the win flag.

Game $\mathcal{G}_{\text{GC},n}^{\text{INT-}x}(\mathcal{A}_{\text{INT-}x})$	Game $\mathcal{G}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}})$
$\text{win} \leftarrow \text{false}; (st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC.Init}(n)$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $S \leftarrow \emptyset; \mathcal{A}_{\text{INT-}x}^{\text{SEND,RECV}}; \text{Return win}$ $\text{SEND}(i, m) \quad // i \in [n], m \in \text{GC.MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1; (st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$ $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\} \quad // \text{INT-PCTXT}$ $S \leftarrow S \cup \{(i, j, c) \mid j \in [n] \setminus \{i\}\} \quad // \text{INT-CTXT}$ $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m) \mid j \in [n] \setminus \{i\}\} \quad // \text{INT-PTXT}$ $\text{Return } (\mathcal{G}_i^l, c)$ $\text{RECV}(i, j, c) \quad // i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}) \leftarrow \text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ $\text{If } m = \perp \text{ then return } \perp$ $\text{entry} \leftarrow (i, j, \text{ctr}, m, c) \quad // \text{INT-PCTXT}$ $\text{entry} \leftarrow (i, j, c) \quad // \text{INT-CTXT}$ $\text{entry} \leftarrow (i, j, \text{ctr}, m) \quad // \text{INT-PTXT}$ $\text{If entry} \notin S \text{ then win} \leftarrow \text{true else } S \leftarrow S \setminus \{\text{entry}\}$ $\text{Return } (\mathcal{G}_j^l, m, \text{ctr})$	$\text{win} \leftarrow \text{false}; (st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC.Init}(n)$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $\mathcal{G} \leftarrow \text{InitGraph}(n); \mathcal{A}_{\text{CA}}^{\text{SEND,RECV}}; \text{Return win}$ $\text{SEND}(i, m) \quad // i \in [n], m \in \text{GC.MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ $\text{Return } (\mathcal{G}_i^l, c)$ $\text{RECV}(i, j, c) \quad // i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}) \leftarrow \text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ $\text{If } m = \perp \text{ then return } \perp$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$ $\text{If } \text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j) \text{ then win} \leftarrow \text{true}$ $\text{Return } (\mathcal{G}_j^l, m, \text{ctr})$

Fig. 11: **Left pane:** Game $\mathcal{G}^{\text{INT-}x}$ defining integrity of GC with parameters n and $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$. **Right pane:** Game $\mathcal{G}^{\text{CA}}$ defining causal awareness of GC with parameters n , restrict , and redact .

The SEND oracle takes a sender i and message m , increments ctr_i , runs GC.Send , and returns $(\mathcal{G}_i^l, c)$. For each $j \in [n] \setminus \{i\}$ it stores in S the tuple $(i, j, \text{ctr}_i, m, c)$, (i, j, c) , or (i, j, ctr_i, m) for INT-PCTXT, INT-CTXT, or INT-PTXT, respectively. In the INT-CTXT game, S is a multiset.

The RECV oracle takes (i, j, c) and runs the algorithm $\text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ to obtain $(st_j, \mathcal{G}_j^l, m, \text{ctr})$. If $m \neq \perp$, it checks whether the corresponding tuple belongs to S , removes it if so, sets win otherwise, and returns $(\mathcal{G}_j^l, m, \text{ctr})$.

6.2 Causal awareness

Causal awareness captures the intuition that an adversary should not be able to cause a receiver to accept a ciphertext out of its causal context. It is analogous to graph correctness, except that, instead of requiring the adversary to forward honestly generated ciphertexts to receivers, it allows the adversary to send arbitrary ciphertexts to the group.

We formally define the causal awareness security of a group chat GC among n users with respect to a restricted graph function restrict and redaction function redact in Fig. 11. The advantage of adversary $\mathcal{A}_{\text{CA}}$ against the CA security of GC with respect to restrict and redact is given by $\text{Adv}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}}) = \Pr[\mathcal{G}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}})]$. The adversary is given access to the SEND and RECV oracles and is tasked with setting the win flag.

The SEND oracle takes (i, m) , increments the sender's sending index ctr_i , and runs $\text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$ to obtain $(st_i, \mathcal{G}_i^l, c)$. It records the current send event by running $\text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ and returns $(\mathcal{G}_i^l, c)$.

The RECV oracle takes (i, j, c) and runs $\text{GC.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ to obtain $(st_j, \mathcal{G}_j^l, m, \text{ctr})$. If $m = \perp$, it returns $\perp$; otherwise, it records the receive event using $\text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$, sets win if $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$, and returns $(\mathcal{G}_j^l, m, \text{ctr})$. The purpose and variants of the restricted graph and redaction functions used in $\mathcal{G}^{\text{CA}}$ are the same as discussed in Section 4.1.

The following result shows that the causal awareness of a group chat follows from its INT-PCTXT integrity and graph correctness.

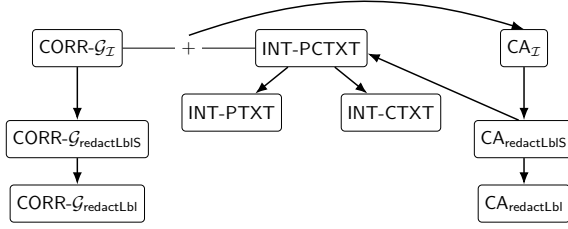


Fig. 12: Relations between correctness, integrity, and causal awareness notions for group chats.

Theorem 5. Let GC be a group chat. Let $n \in \mathbb{N}$ with $n \geq 1$. Let redact be a redaction function and restrict a restricted graph function. Let $\mathcal{A}_{CA}$ be an adversary against the CA security of GC with respect to n , restrict , and redact . Then we build adversaries $\mathcal{B}_{\text{INT-PCTXT}}$ and $\mathcal{B}_{\text{CORR-G}}$ such that

$$\text{Adv}_{GC,n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{CA}) \leq \text{Adv}_{GC,n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}}) + \text{Adv}_{GC,n,\text{restrict},\text{redact}}^{\text{CORR-G}}(\mathcal{B}_{\text{CORR-G}}).$$

Proof sketch. We prove Theorem 5 via hybrids starting with $\mathcal{G}_{GC,n,\text{restrict},\text{redact}}^{\text{CA}}$. The first hop prevents $\mathcal{A}_{CA}$ from winning through an oracle RECV query for which the algorithm $GC.\text{Recv}$ accepts a ciphertext not attributable to an honest send; this hop is bounded by INT-PCTXT security. The second hop removes the win condition triggered when the receiver's redacted local graph differs from that receiver's restriction of the redacted global graph; this hop is bounded by CORR-G security. The final hybrid cannot be won. The formal proof appears in Appendix A.4.

Relations between notions. Fig. 12 summarizes relations between our notions of correctness, integrity, and causal awareness. First, Theorem 5 establishes the main connection between graph correctness, INT-PCTXT integrity, and causal awareness.

Second, for each fixed $\text{restrict} \in \{\text{rGMax}, \text{rG2Hop}\}$, agreement on a given set of label fields implies agreement on any subset of those fields. This gives the downward implications within both the correctness column (left) and the causal-awareness column (right) of Fig. 12.

Third, INT-PCTXT requires an accepted receive to match an honest send on the full record (i, j, ctr, m, c) , whereas INT-PTXT and INT-CTXT require a match only on part of it, the plaintext fields (i, j, ctr, m) or the ciphertext (i, j, c) . Matching on all fields implies matching on a subset of fields. This gives $\text{INT-PCTXT} \Rightarrow \text{INT-CTXT}$ and $\text{INT-PCTXT} \Rightarrow \text{INT-PTXT}$.

Finally, under redactLblS , all of a receiver's own events retain their (m, c) labels. Thus, $\text{CA}_{\text{redactLblS}}$ requires the receiver's reconstructed graph to agree with the global graph on those labeled events, whereas INT-PCTXT requires only that each accepted (m, c) pair come from an honest send. Agreement on a receive event therefore implies that the corresponding message was honestly sent. This gives $\text{CA}_{\text{redactLblS}} \Rightarrow \text{INT-PCTXT}$ for all channels whose $GC.\text{Recv}$ adds $(j, \text{RECV}, \text{ctr}, (m, c))$ to receiver j 's local graph upon accepting c and returning (m, ctr) .

6.3 Constructing causally aware group chats

We construct the group chat GtE (Graph-then-Encrypt) by composing the unencrypted group chat GC_G with the encrypted bidirectional channel GC_E . Our GtE construction is shown in Fig. 13.

Graph correctness of GtE under redactLbl follows from graph correctness of GC_G and message correctness of GC_E (Theorem 6). Its INT-PCTXT integrity follows from INT-PCTXT integrity of GC_E and message correctness of GC_G (Theorem 7). Our general result then gives causal awareness of GtE .

Both layers initialize the same n users, and each payload returned by $GC_G.\text{Send}$ must belong to $GC_E.\text{MS}$. We view GC_E in the group-chat syntax by fixing its local graphs to $\perp$, set $\text{GtE}.\text{MS} := GC_G.\text{MS}$, and use the product of the component states.

$\text{GtE}.\text{Recv}$ commits its component states only if both component calls accept; otherwise it returns the input state and local graph with $(m, \text{ctr}) = (\perp, \perp)$.

GtE.Init(n) $(st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l) \leftarrow \$ GC_G.\text{Init}(n)$ $(st_1^E, \perp), \dots, (st_n^E, \perp) \leftarrow \$ GC_E.\text{Init}(n)$ $st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$ Return $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l)$	GtE.Recv($j, st_j, \mathcal{G}_j^l, i, c$) $(st_j^G, st_j^E) \leftarrow st_j$ $(m, \text{ctr}) \leftarrow (\perp, \perp)$ $(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow GC_E.\text{Recv}(j, st_j^E, \perp, i, c)$ If $\text{payload} \neq \perp$ then $(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow GC_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$ If $m \neq \perp$ then $st_j \leftarrow (st_j^G, st_j^E); \text{ctr} \leftarrow \text{ctr}_E$ Return $(st_j, \mathcal{G}_j^l, m, \text{ctr})$
GtE.Send($i, st_i, \mathcal{G}_i^l, m$) $(st_i^G, st_i^E) \leftarrow st_i$ $(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \$ GC_G.\text{Send}(i, st_i^G, \mathcal{G}_i^l, m)$ $(st_i^E, \perp, c) \leftarrow \$ GC_E.\text{Send}(i, st_i^E, \perp, \text{payload})$ $st_i \leftarrow (st_i^G, st_i^E);$ Return $(st_i, \mathcal{G}_i^l, c)$	

Fig. 13: Group chat GtE = GRAPH-THEN-ENCRYPT-GC[GC_G, GC_E].

Construction 1. Let GC_G be a group chat. Let GC_E be a bidirectional channel. Then the group chat GtE = GRAPH-THEN-ENCRYPT-GC[GC_G, GC_E] is as defined in Fig. 13.

Theorem 6. Let GtE be as in Construction 1, and let $n \in \mathbb{N}$ with $n \geq 1$. Let restrict be a restricted graph function on views obtained by applying redactLbl to causality graphs. Let $\mathcal{A}_{\text{CORR-G}}$ be an adversary against the CORR-G security of GtE with respect to n , restrict, and redactLbl. Then we build adversaries $\mathcal{B}_{\text{CORR-m}}$ and $\mathcal{B}_{\text{CORR-G}}$ such that

$$\text{Adv}_{\text{GtE}, n, \text{restrict}, \text{redactLbl}}^{\text{CORR-G}}(\mathcal{A}_{\text{CORR-G}}) \leq \text{Adv}_{\text{GC}_G, n}^{\text{CORR-m}}(\mathcal{B}_{\text{CORR-m}}) + \text{Adv}_{\text{GC}_G, n, \text{restrict}, \text{redactLbl}}^{\text{CORR-G}}(\mathcal{B}_{\text{CORR-G}}).$$

Proof sketch. We modify the graph-correctness game of GtE to abort unless the payload and counter returned by GC_E.Recv match the selected honest GC_E.Send call; this hop is bounded by message correctness of GC_E. Thereafter, the global graph and the graph in GC_G's correctness game differ only in their label fields. Applying redactLbl before restrict reduces a graph-correctness violation for GtE to one for GC_G. The formal proof appears in Appendix A.5.

Theorem 7. Let GtE be as in Construction 1, and let $n \in \mathbb{N}$ with $n \geq 1$. Let $\mathcal{A}_{\text{INT-PCTXT}}$ be an adversary against the INT-PCTXT security of GtE with respect to n . Then we build adversaries $\mathcal{B}_{\text{CORR-m}}$ and $\mathcal{B}_{\text{INT-PCTXT}}$ such that

$$\text{Adv}_{\text{GtE}, n}^{\text{INT-PCTXT}}(\mathcal{A}_{\text{INT-PCTXT}}) \leq \text{Adv}_{\text{GC}_G, n}^{\text{CORR-m}}(\mathcal{B}_{\text{CORR-m}}) + \text{Adv}_{\text{GC}_E, n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}}).$$

Proof sketch. First consider an encryption-layer output identifying an honest GC_G.Send call for which GC_G.Recv returns a different message, including $m = \perp$; message correctness of GC_G bounds this event. Once excluded, INT-PCTXT security of GC_E bounds an accepting GC_E.Recv call with no matching honest GC_E.Send entry. Otherwise entry = (i, j, ctr, m, c) belongs to S , so $\mathcal{A}_{\text{INT-PCTXT}}$ cannot win on that query. The formal proof appears in Appendix A.6.

7 Checkpoint-aided causal awareness

Checkpoint-aided causal awareness reduces per-message overhead by moving causal catch-up work from ordinary messages to explicit epoch-transition objects. This mirrors the MLS split between application messages sent within an epoch and Commit-like objects that advance the group to a new epoch. A checkpoint carries the information needed to close the causal view for the old epoch; to enter the next epoch, a receiver must apply the checkpoint that defines it.

Intuitively, a checkpoint is a settlement point. Without checkpoints, a sender may need to attach old causal information to many later messages, especially for recipients that were offline or missed prior messages. With checkpoints, this old information is reconciled once at an epoch boundary; messages in the next epoch can refer to the new epoch state instead of re-carrying the same history.

$$\begin{aligned}
(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) &\leftarrow \text{GC}_{\text{CP}}.\text{Init}(n) \\
(st_i, \mathcal{G}_i^l, c) &\leftarrow \text{GC}_{\text{CP}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m) \\
(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}^{\text{in-epoch}}) &\leftarrow \text{GC}_{\text{CP}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c) \\
(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}) &\leftarrow \text{GC}_{\text{CP}}.\text{CreateChkPt}(i, st_i, \mathcal{G}_i^l) \\
(st_j, \mathcal{G}_j^l, \text{ctr}^{\text{in-epoch}}, \text{th}^{\text{new}}) &\leftarrow \text{GC}_{\text{CP}}.\text{ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)
\end{aligned}$$

Fig. 14: Syntax of a group chat with checkpoints.

Checkpoint syntax. The syntax is shown in Fig. 14. It extends a group chat. Recv now returns the transcript hash that identifies the epoch of the received message and the message’s within-epoch action counter. CreateChkPt creates a checkpoint ciphertext and returns the transcript hash of the new epoch. ApplyChkPt processes a checkpoint ciphertext and returns the within-epoch action counter of that ciphertext together with the new transcript hash it derives. ApplyChkPt is deterministic and state-preserving when it returns $\text{th}^{\text{new}} = \perp$. A checkpoint creator may later apply the selected checkpoint itself ($i = j$).

Bookkeeping conventions. The games use two counters. The counter $\text{ctr}_j^{\text{game}}$ is a game-only event counter indexing honest message sends and checkpoint creations. The counter $\text{ctr}_j^{\text{in-epoch}}$ is the logical action counter in j ’s current epoch th_j . Together with sender i , $(\text{th}_i, \text{ctr}_i^{\text{in-epoch}})$ identifies the send vertex for both messages and checkpoints. Following MLS, transcript hashes are epoch labels; we do not model them as secrets. The table $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}]$ stores the record that the games later use to check Recv and ApplyChkPt outputs. The graph uses a related label (τ, y, c) , where $\tau \in \{\text{MSG}, \text{CHKPT}\}$, y is a message or new transcript hash, and c is the transport object.

Checkpoint correctness. The message-correctness game is given in Fig. 15. It checks that Recv returns the message, epoch, and action counter associated with the corresponding honest Send call, and that ApplyChkPt returns the action counter and child hash associated with the corresponding honest CreateChkPt call. Unlike the checkpoint graph-correctness game, the game in Fig. 15 neither initializes nor updates a global causality graph and does not compare the receiver’s redacted local graph with a restriction of the redacted global graph.

For a query to the oracle SEND, the checkpoint message-correctness game adds only the sent message m to the event-information tuple used by graph correctness. The added check after the algorithm $\text{GC}_{\text{CP}}.\text{Recv}$ compares the returned message, epoch, and action counter with the recorded values. The added check after the algorithm $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$ similarly compares the returned action counter and child transcript hash with the recorded values. Any mismatch, including rejection, sets win. We state graph correctness in Fig. 16: successful calls to the oracles RECV and APPLYCHKPT must update local graphs consistently with the required view of the redacted global graph. The set openEpochs_j records the epoch labels for which j is treated as holding receive secrets. The set usedEpochs_j records transcript hashes already used in j ’s chain, so the game can require that this chain contains no repeated transcript hash. Here $\text{th}_{\text{ref}}^{\text{new}}$ is the child hash recorded at checkpoint creation, which is used for reuse detection and epoch bookkeeping.

Checkpoint integrity. The integrity game is shown in Fig. 16. It is parameterized by $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$. Variant-specific lines are marked INT-PCTXT-chkpt, INT-CTXT-chkpt, or INT-PTXT-chkpt. In each group of such lines, only the line marked with the selected variant is active. That line says what must match an honest send or checkpoint output: the record and ciphertext, the ciphertext alone, or the record alone. In the INT-CTXT-chkpt variant, the matched record is only (i, j, c) . Since the channel is stateful, this variant authenticates the ciphertext as an honest transport object but does not fix the epoch, counter, or message to which a valid ciphertext decrypts. The message-receive oracle also checks openEpochs_j : a successful decryption must report an epoch for which j already has receive secrets.

Checkpoint causal awareness. The causal-awareness game is shown in Fig. 17. It allows the adversary to choose message and checkpoint ciphertexts directly. The notion is deliberately permissive: every successful processing step is judged only by whether the receiver’s redacted local graph matches the required restriction of the redacted global graph after adding the corresponding receive vertex.

Game $\mathcal{G}_{\text{GC}_{\text{CP}},n}^{\text{CORR-}m\text{-chkpt}}(\mathcal{A}_{\text{CORR-}m\text{-chkpt}})$ win $\leftarrow$ false $(\text{ctr}_j^{\text{game}}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow (0, 0) \quad \forall j \in [n]$ For all $j \in [n]$ do $(\text{th}_j, \text{openEpochs}_j, \text{usedEpochs}_j) \leftarrow (\epsilon, \{\epsilon\}, \{\epsilon\})$ $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{CP}}.\text{Init}(n)$ $\mathcal{A}_{\text{CORR-}m\text{-chkpt}}^{\text{O}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ Return win <u>SEND(i, m)</u> // $i \in [n], m \in \text{GC}_{\text{CP}}.\text{MS}$ $\text{ctr}_i^{\text{game}} \leftarrow \text{ctr}_i^{\text{game}} + 1$; $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC}_{\text{CP}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \leftarrow (\text{MSG}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, m)$ $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$ Return $(st_i, \mathcal{G}_i^l, c, \text{ctr}_i^{\text{game}})$ <u>CREATECHKPT(i)</u> // $i \in [n]$ $\text{ctr}_i^{\text{game}} \leftarrow \text{ctr}_i^{\text{game}} + 1$; $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}) \leftarrow \text{GC}_{\text{CP}}.\text{CreateChkPt}(i, st_i, \mathcal{G}_i^l)$ $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \leftarrow (\text{CHKPT}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, \text{th}^{\text{new}})$ $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow c \quad \forall j \in [n]$ Return $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}, \text{ctr}_i^{\text{game}})$	<u>RECV($i, j, \text{ctr}_j^{\text{game}}$)</u> // $i, j \in [n], i \neq j, \text{ctr}_j^{\text{game}} \in \mathbb{N}$ require $C[i, j, \text{ctr}_j^{\text{game}}] \neq \perp$ require $\text{eventInfo}[i, \text{ctr}_j^{\text{game}}] \in S$ $c \leftarrow C[i, j, \text{ctr}_j^{\text{game}}]$ $(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow \text{GC}_{\text{CP}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{eventInfo}[i, \text{ctr}_j^{\text{game}}] \neq (\text{MSG}, \text{th}, \text{ctr}_j^{\text{in-epoch}}, m)$ then win $\leftarrow$ true $C[i, j, \text{ctr}_j^{\text{game}}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}})$ <u>APPLYCHKPT($i, j, \text{ctr}_j^{\text{game}}$)</u> // $i, j \in [n], \text{ctr}_j^{\text{game}} \in \mathbb{N}$ require $C[i, j, \text{ctr}_j^{\text{game}}] \neq \perp$ require $\text{eventInfo}[i, \text{ctr}_j^{\text{game}}] \in \{\text{CHKPT}\} \times \{\text{th}_j\} \times \mathbb{N} \times \{0, 1\}^*$ $c \leftarrow C[i, j, \text{ctr}_j^{\text{game}}]$ $(\text{CHKPT}, \text{th}_{\text{ref}}, \text{ctr}_{\text{ref}}^{\text{in-epoch}}, \text{th}_{\text{ref}}^{\text{new}}) \leftarrow \text{eventInfo}[i, \text{ctr}_j^{\text{game}}]$ $(st_j, \mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}_{\text{ref}}^{\text{new}}) \leftarrow \text{GC}_{\text{CP}}.\text{ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{eventInfo}[i, \text{ctr}_j^{\text{game}}] \neq (\text{CHKPT}, \text{th}_j, \text{ctr}_j^{\text{in-epoch}}, \text{th}_{\text{ref}}^{\text{new}})$ then win $\leftarrow$ true If $\text{th}_{\text{ref}}^{\text{new}} \in \text{usedEpochs}_j$ then win $\leftarrow$ true $\text{th}_j \leftarrow \text{th}_{\text{ref}}^{\text{new}}$; $\text{openEpochs}_j \leftarrow \text{openEpochs}_j \cup \{\text{th}_{\text{ref}}^{\text{new}}\}$ $\text{usedEpochs}_j \leftarrow \text{usedEpochs}_j \cup \{\text{th}_{\text{ref}}^{\text{new}}\}$; $\text{ctr}_j^{\text{in-epoch}} \leftarrow 0$ $C[i, j, \text{ctr}_j^{\text{game}}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}_{\text{ref}}^{\text{new}})$
---	---

Fig. 15: Game $\mathcal{G}^{\text{CORR-}m\text{-chkpt}}$ defining message correctness of GC_{CP} with parameter n . Here $\text{O} = \{\text{SEND}, \text{RECV}, \text{CREATECHKPT}, \text{APPLYCHKPT}\}$ and $S = \{\text{MSG}\} \times \text{openEpochs}_j \times \mathbb{N} \times \text{GC}_{\text{CP}}.\text{MS}$.

Theorem 8. Let GC_{CP} be a group chat with checkpoints. Let $n \in \mathbb{N}$ with $n \geq 1$. Let *redact* be a redaction function and *restrict* a restricted graph function on graph views. Let $\mathcal{A}_{\text{CA-chkpt}}$ be an adversary against the CA-chkpt security of GC_{CP} with respect to n , *restrict*, and *redact*. Then we build adversaries $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ and $\mathcal{B}_{\text{CORR-}g\text{-chkpt}}$ such that

$$\text{Adv}_{\text{GC}_{\text{CP}},n,\text{restrict},\text{redact}}^{\text{CA-chkpt}}(\mathcal{A}_{\text{CA-chkpt}}) \leq \text{Adv}_{\text{GC}_{\text{CP}},n}^{\text{INT-PCTXT-chkpt}}(\mathcal{B}_{\text{INT-PCTXT-chkpt}}) + \text{Adv}_{\text{GC}_{\text{CP}},n,\text{restrict},\text{redact}}^{\text{CORR-}g\text{-chkpt}}(\mathcal{B}_{\text{CORR-}g\text{-chkpt}}).$$

The proof and reduction games are in [Appendix B](#).

8 Server-aided causal awareness

In the server-aided setting, message ciphertexts and causal evidence are delivered separately. A sender gives the server both an in-band ciphertext and a corresponding evidence object. The server can fan out the ciphertext immediately and store the evidence for later retrieval. A receiver can display the message immediately and, if needed, fetch the evidence later to reconstruct the sender's causal context.

The syntax in [Fig. 18](#) makes this split explicit: *Send* returns an in-band ciphertext c and delayed evidence *evidence*, *Recv* returns the message, its send-counter value, and a verification token, and *Verify* checks candidate evidence against that token. Thus, a receiver need not download the evidence merely to display the message. If every receiver later fetches every evidence object, however, the asymptotic worst-case communication cost is unchanged.

The split limits availability: the delivery service can withhold evidence and thereby delay causal awareness. Withholding alone cannot create accepted evidence. It also requires evidence binding: each ciphertext must bind its delayed evidence so that the service cannot substitute a different evidence object. A construction can authenticate a commitment inside the encrypted plaintext or as authenticated sidecar data; encrypting the evidence itself is optional. The proof appears in [Appendix C](#).

We now formalize the server-aided setting and state its central implication: graph correctness and INT-PCTXT security imply causal awareness. The proof appears in [Appendix C](#).

Game $\mathcal{G}_{\text{GCP},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}\text{-chkpt}}(\mathcal{A}_{\text{CORR-}\mathcal{G}\text{-chkpt}})$ win $\leftarrow$ false ; $\mathcal{G} \leftarrow \text{InitGraph}(n)$ $(\text{ctr}_j^{\text{game}}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow (0, 0) \quad \forall j \in [n]$ $(\text{th}_j, \text{openEpochs}_j, \text{usedEpochs}_j) \leftarrow (\epsilon, \{\epsilon\}, \{\epsilon\}) \quad \forall j \in [n]$ $(st_1, \mathcal{G}_1^l, \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GCCP.Init}(n)$ $\mathcal{A}_{\text{CORR-}\mathcal{G}\text{-chkpt}}^{\text{SEND,RECV,CREATECHKPT,APPLYCHKPT}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ Return win SEND(i, m) // $i \in [n], m \in \text{GCCP.MS}$ $\text{ctr}_i^{\text{game}} \leftarrow \text{ctr}_i^{\text{game}} + 1$; $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GCCP.Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{MSG}, m, c))$ $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \leftarrow (\text{MSG}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}})$ $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$ Return $(st_i, \mathcal{G}_i^l, c, \text{ctr}_i^{\text{game}})$ CREATECHKPT(i) // $i \in [n]$ $\text{ctr}_i^{\text{game}} \leftarrow \text{ctr}_i^{\text{game}} + 1$; $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}) \leftarrow \text{GCCP.CreateChkPt}(i, st_i, \mathcal{G}_i^l)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{CHKPT}, \text{th}^{\text{new}}, c))$ $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \leftarrow (\text{CHKPT}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, \text{th}^{\text{new}})$ $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow c \quad \forall j \in [n]$ Return $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}, \text{ctr}_i^{\text{game}})$	RECV($i, j, \text{ctr}_i^{\text{game}}$) // $i, j \in [n], i \neq j, \text{ctr}_i^{\text{game}} \in \mathbb{N}$ require $C[i, j, \text{ctr}_i^{\text{game}}] \neq \perp$ require $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \in \{\text{MSG}\} \times \text{openEpochs}_j \times \mathbb{N}$ $c \leftarrow C[i, j, \text{ctr}_i^{\text{game}}]$; $(\text{MSG}, \text{th}_{\text{ref}}, \text{ctr}_{\text{ref}}^{\text{in-epoch}}) \leftarrow \text{eventInfo}[i, \text{ctr}_i^{\text{game}}]$ $(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow \text{GCCP.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}_{\text{ref}}, \text{ctr}_{\text{ref}}^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then win $\leftarrow$ true $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}})$ APPLYCHKPT($i, j, \text{ctr}_i^{\text{game}}$) // $i, j \in [n], \text{ctr}_i^{\text{game}} \in \mathbb{N}$ require $C[i, j, \text{ctr}_i^{\text{game}}] \neq \perp$ require $\text{eventInfo}[i, \text{ctr}_i^{\text{game}}] \in \{\text{CHKPT}\} \times \{\text{th}_j\} \times \mathbb{N} \times \{0, 1\}^*$ $c \leftarrow C[i, j, \text{ctr}_i^{\text{game}}]$ $(\text{CHKPT}, \text{th}_{\text{ref}}, \text{ctr}_{\text{ref}}^{\text{in-epoch}}, \text{th}^{\text{new}}) \leftarrow \text{eventInfo}[i, \text{ctr}_i^{\text{game}}]$ $(st_j, \mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}) \leftarrow \text{GCCP.ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{th}^{\text{new}} = \perp$ then return $\perp$ If $\text{th}_{\text{ref}}^{\text{new}} \in \text{usedEpochs}_j$ then win $\leftarrow$ true $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}_{\text{ref}}, \text{ctr}_{\text{ref}}^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then win $\leftarrow$ true $\text{th}_j \leftarrow \text{th}_{\text{ref}}^{\text{new}}$; $\text{openEpochs}_j \leftarrow \text{openEpochs}_j \cup \{\text{th}_{\text{ref}}^{\text{new}}\}$ $\text{usedEpochs}_j \leftarrow \text{usedEpochs}_j \cup \{\text{th}_{\text{ref}}^{\text{new}}\}$; $\text{ctr}_j^{\text{in-epoch}} \leftarrow 0$ $C[i, j, \text{ctr}_i^{\text{game}}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$
Game $\mathcal{G}_{\text{GCCP},n}^{\text{INT-x-chkpt}}(\mathcal{A}_{\text{INT-x-chkpt}})$ win $\leftarrow$ false ; $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GCCP.Init}(n)$ $(\text{ctr}_j^{\text{in-epoch}}, \text{th}_j, \text{openEpochs}_j) \leftarrow (0, \epsilon, \{\epsilon\}) \quad \forall j \in [n]$ $S \leftarrow \emptyset$; $\mathcal{A}_{\text{INT-x-chkpt}}^{\text{SEND,RECV,CREATECHKPT,APPLYCHKPT}}$; Return win SEND(i, m) // $i \in [n], m \in \text{GCCP.MS}$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GCCP.Send}(i, st_i, \mathcal{G}_i^l, m)$ $\text{msgInfo} \leftarrow (\text{MSG}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, m)$ $S \leftarrow S \cup \{(i, j, \text{msgInfo}, c) \mid j \in [n] \setminus \{i\}\}$ // INT-PCTXT-chkpt $S \leftarrow S \cup \{(i, j, c) \mid j \in [n] \setminus \{i\}\}$ // INT-CTXT-chkpt $S \leftarrow S \cup \{(i, j, \text{msgInfo}) \mid j \in [n] \setminus \{i\}\}$ // INT-PTXT-chkpt Return $(\mathcal{G}_i^l, c)$ CREATECHKPT(i) // $i \in [n]$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}) \leftarrow \text{GCCP.CreateChkPt}(i, st_i, \mathcal{G}_i^l)$ $\text{chkptInfo} \leftarrow (\text{CHKPT}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, \text{th}^{\text{new}})$ $S \leftarrow S \cup \{(i, j, \text{chkptInfo}, c) \mid j \in [n]\}$ // INT-PCTXT-chkpt $S \leftarrow S \cup \{(i, j, c) \mid j \in [n]\}$ // INT-CTXT-chkpt $S \leftarrow S \cup \{(i, j, \text{chkptInfo}) \mid j \in [n]\}$ // INT-PTXT-chkpt Return $(\mathcal{G}_i^l, c, \text{th}^{\text{new}})$	RECV(i, j, c) // $i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow \text{GCCP.Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ If $\text{th} \notin \text{openEpochs}_j$ then win $\leftarrow$ true $\text{msgInfo} \leftarrow (\text{MSG}, \text{th}, \text{ctr}_j^{\text{in-epoch}}, m)$ entry $\leftarrow (i, j, \text{msgInfo}, c)$ // INT-PCTXT-chkpt entry $\leftarrow (i, j, c)$ // INT-CTXT-chkpt entry $\leftarrow (i, j, \text{msgInfo})$ // INT-PTXT-chkpt If entry $\notin S$ then win $\leftarrow$ true else $S \leftarrow S \cup \{\text{entry}\}$ Return $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}})$ APPLYCHKPT(i, j, c) // $i, j \in [n], c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}) \leftarrow \text{GCCP.ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{th}^{\text{new}} = \perp$ then return $\perp$ $\text{chkptInfo} \leftarrow (\text{CHKPT}, \text{th}_j, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$ entry $\leftarrow (i, j, \text{chkptInfo}, c)$ // INT-PCTXT-chkpt entry $\leftarrow (i, j, c)$ // INT-CTXT-chkpt entry $\leftarrow (i, j, \text{chkptInfo})$ // INT-PTXT-chkpt If entry $\notin S$ then win $\leftarrow$ true else $S \leftarrow S \cup \{\text{entry}\}$ $\text{th}_j \leftarrow \text{th}^{\text{new}}$; $\text{openEpochs}_j \leftarrow \text{openEpochs}_j \cup \{\text{th}^{\text{new}}\}$ $\text{ctr}_j^{\text{in-epoch}} \leftarrow 0$; Return $(\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$

Fig. 16: **Top pane:** Game $\mathcal{G}^{\text{CORR-}\mathcal{G}\text{-chkpt}}$ defining graph correctness of GCCP with parameters n , restrict, and redact. **Bottom pane:** Game $\mathcal{G}^{\text{INT-x-chkpt}}$ defining integrity of GCCP with parameters n and $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$.

Server-aided syntax. Fig. 18 gives the interface; Verify is deterministic and Boolean-valued and, on rejection, leaves the state and local graph unchanged.

Delayed-evidence graph functions. An accepted Recv adds an ordinary receive vertex; a later accepted Verify invokes AddEvidenceReceipt to append an evidence receipt. Fig. 19 declares the inherited base functions and defines AddEvidenceReceipt.

In a server-aided group chat, the algorithm Recv can return a message before its causal evidence is fetched and verified. An ordinary receive vertex records this delivery. A later successful verification is

Game $\mathcal{G}_{\text{GC}_{\text{CP}}, n, \text{restrict}, \text{redact}}^{\text{CA-chkpt}}(\mathcal{A}_{\text{CA-chkpt}})$ win $\leftarrow$ false ; $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{CP}}.\text{Init}(n)$ $(\text{ctr}_j^{\text{in-epoch}}, \text{th}_j) \leftarrow (0, \epsilon) \quad \forall j \in [n]$ $\mathcal{G} \leftarrow \text{InitGraph}(n)$; $\mathcal{A}_{\text{CA-chkpt}}^{\text{SEND, RECV, CREATECHKPT, APPLYCHKPT}}$ Return win <u>SEND(i, m)</u> // $i \in [n], m \in \text{GC}_{\text{CP}}.\text{MS}$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC}_{\text{CP}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{MSG}, m, c))$ Return $(\mathcal{G}_i^l, c)$ <u>CREATECHKPT(i)</u> // $i \in [n]$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}) \leftarrow \text{GC}_{\text{CP}}.\text{CreateChkPt}(i, st_i, \mathcal{G}_i^l)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{CHKPT}, \text{th}^{\text{new}}, c))$ Return $(\mathcal{G}_i^l, c, \text{th}^{\text{new}})$	<u>RECV(i, j, c)</u> // $i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}) \leftarrow \text{GC}_{\text{CP}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}, \text{ctr}_j^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then win $\leftarrow$ true Return $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}})$ <u>APPLYCHKPT(i, j, c)</u> // $i, j \in [n], c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}) \leftarrow \text{GC}_{\text{CP}}.\text{ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{th}^{\text{new}} = \perp$ then return $\perp$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}_j, \text{ctr}_j^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then win $\leftarrow$ true $\text{th}_j \leftarrow \text{th}^{\text{new}} ; \text{ctr}_j^{\text{in-epoch}} \leftarrow 0$ Return $(\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$
--	--

Fig. 17: Game $\mathcal{G}^{\text{CA-chkpt}}$ defining causal awareness of GC_{CP} with parameters n , restrict, and redact.

$(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$
 $(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$
 $(st_j, \mathcal{G}_j^l, m, \text{ctr}, \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$
 $(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence})$

Fig. 18: Syntax of a server-aided group chat.

InitGraph(n)
nextCtr(V)
AddSendVrtx($\mathcal{G}, i, \text{eid}, \text{label}$)
AddRecvVrtx($\mathcal{G}, i, j, \text{eid}$)
// All four functions are as in Fig. 3.
AddEvidenceReceipt($\mathcal{G}, i, j, \text{eid}$) // $i, j \in [n]$
 $(n, V_1, \dots, V_n, E) \leftarrow \mathcal{G}$
 $U \leftarrow \{u \in V_i \mid u.\text{type} = \text{SEND} \wedge u.\text{eid} = \text{eid}\}$
If $|U| \neq 1$ then return $\mathcal{G}$
 $\{u\} \leftarrow U ; R \leftarrow \{v \in V_j \mid (u, v) \in E\}$
 $R \leftarrow \{v \in R \mid v.\text{type} = \text{RECV} \wedge v.\text{eid} = \text{eid}\}$
If $|R| \neq 1$ then return $\mathcal{G}$
 $T \leftarrow \{r \in V_j \mid r.\text{type} = \text{RECEIPT} \wedge r.\text{eid} = (i, \text{eid})\}$
If $|T| \neq 0$ then return $\mathcal{G}$
 $\text{ctr} \leftarrow \text{nextCtr}(V_j) ; r \leftarrow (j, \text{RECEIPT}, \text{ctr}, (i, \text{eid}), \text{true})$
 $V_j \leftarrow V_j \cup \{r\} ; \text{Return } (n, V_1, \dots, V_n, E)$

Fig. 19: Delayed-evidence graph functions.

recorded separately by appending a matching evidence-receipt vertex at the receiver's next local counter; rejected verification adds no event. This records the order of delivery and verification without modifying the earlier delivery record. The functions `InitGraph`, `nextCtr`, `AddSendVrtx`, and `AddRecvVrtx` are unchanged from Fig. 3. Fig. 19 declares these inherited functions by reference and defines the new `AddEvidenceReceipt` function.

Game $\mathcal{G}_{\text{GC}_{\text{SA}},n}^{\text{CORR-}m\text{-SA}}(\mathcal{A}_{\text{CORR-}m\text{-SA}})$	$\text{SEND}(i, m) \quad // \ i \in [n], m \in \text{GC}_{\text{SA}}.\text{MS}$
$\text{win} \leftarrow \text{false}$	$\text{ctr}_i \leftarrow \text{ctr}_i + 1$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$
$(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$	$\text{sendEvidence}[i, \text{ctr}_i] \leftarrow \text{evidence}$
$\mathcal{A}_{\text{CORR-}m\text{-SA}}^{\text{SEND,RECV,VERIFY}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l)); \text{Return win}$	$M[i, \text{ctr}_i] \leftarrow m$
$\text{VERIFY}(i, j, \text{ctr}) \quad // \ i, j \in [n], \text{ctr} \in \mathbb{N}$	$C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$
$\text{require pendingToken}[i, j, \text{ctr}] \neq \perp$	$\text{Return } (st_i, \mathcal{G}_i^l, c, \text{evidence})$
$\text{verTok} \leftarrow \text{pendingToken}[i, j, \text{ctr}]$	$\text{RECV}(i, j, \text{ctr}) \quad // \ i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$
$\text{evidence} \leftarrow \text{sendEvidence}[i, \text{ctr}]$	$\text{require } C[i, j, \text{ctr}] \neq \perp$
$(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence})$	$c \leftarrow C[i, j, \text{ctr}]$
If $\text{acc} = \text{true}$ then	$(st_j, \mathcal{G}_j^l, m, \text{ctr}', \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$
$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \perp$	If $(\text{ctr}', m) \neq (\text{ctr}, M[i, \text{ctr}])$ then $\text{win} \leftarrow \text{true}$
Else $\text{win} \leftarrow \text{true}$	If $m = \perp$ then $\text{return } \perp$
$\text{Return } (\mathcal{G}_j^l, \text{acc})$	$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \text{verTok}$
	$C[i, j, \text{ctr}] \leftarrow \perp; \text{Return } (\mathcal{G}_j^l, m, \text{ctr}')$

Fig. 20: Game $\mathcal{G}^{\text{CORR-}m\text{-SA}}$ defining message correctness of GC_{SA} with parameter n .

In the server-aided setting, a delayed-evidence graph retains the base five-field vertex format, local counter order, and delivery edges, but additionally permits vertices of type **RECEIPT**. Because send-event identifiers are unique only per sender, the receipt vertex for sender i 's event eid uses (i, eid) as its event identifier. The receipt vertex's user field is j , the receiver whose timeline contains the receipt. The event identifier includes the sender because the graph-label field may be removed by redaction. The receipt vertex's graph-label field is the fixed value **true** because **AddEvidenceReceipt** is called only after successful verification. The receipt does not repeat the send label because it represents verification rather than another delivery. This fixed value fills the common graph-label field; no algorithm branches on it. No delivery edge is added for a receipt. The receipt's counter places it after the corresponding receive on the receiver's local timeline without changing that earlier event.

We say a delayed-evidence graph $\mathcal{G}$ is valid if it can be obtained from $\text{InitGraph}(n)$ by a finite sequence of calls to the graph-update functions **AddSendVrtx**, **AddRecvVrtx**, and **AddEvidenceReceipt**. The previously defined local and causal orders, graph restrictions, and redactions apply unchanged.

Server-aided correctness. Message correctness is shown in Fig. 20. It checks that, for an honest in-band ciphertext, **Recv** returns the corresponding message, counter, and verification token, and that **Verify** accepts the evidence produced by the same **Send** call when given that token. Unlike the server-aided graph-correctness game, the game in Fig. 20 neither initializes nor updates a global causality graph and does not compare the receiver's redacted local graph with a required subview of the redacted global graph.

For a query to the oracle **SEND**, the server-aided message-correctness game records the sent message in M under the sender and send counter. The added check after the algorithm $\text{GC}_{\text{SA}}.\text{Recv}$ compares the returned message and send counter with that record. The added check in the oracle **VERIFY** requires the algorithm $\text{GC}_{\text{SA}}.\text{Verify}$ to accept the delayed evidence produced by the honest send identified by (i, ctr) when given the token previously returned by $\text{GC}_{\text{SA}}.\text{Recv}$.

Graph correctness is shown in Fig. 21. The receive oracle records a receive edge in the global graph. The verify oracle records an evidence receipt when honest delayed evidence is accepted. After each update, the receiver's redacted local graph must match the required restriction of the redacted global graph.

Server-aided integrity. The integrity game is shown in Fig. 21. Parameter $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$ selects what an accepted **Recv** on adversarial in-band input must match to an honest send. In each group of variant-specific lines, only the line marked with the selected variant is active. For a token returned by **Recv** on an honest in-band ciphertext, evidence binding requires $\text{Verify}(\text{verTok}, \text{evidence}')$ to accept only the evidence produced by the same **Send** call.

Server-aided causal awareness. The causal-awareness game is shown in Fig. 21. The adversary chooses the in-band component and later chooses candidate evidence. The verification token is returned by the algorithm $\text{GC}_{\text{SA}}.\text{Recv}$ for the delivered in-band component.

The game models outsider-style security: under the server-aided integrity condition used in the theorem, any accepted evidence must originate from an honest send. Modeling insider security would additionally require adversarial senders that can authenticate inconsistent claims; we leave that setting out of scope.

Theorem 9. *Let GC_{SA} be a server-aided group chat. Let $n \in \mathbb{N}$ with $n \geq 1$. Let redact be a redaction function and restrict a restricted graph function on graph views. Let $\mathcal{A}_{\text{CA-SA}}$ be an adversary against the CA-SA security of GC_{SA} with respect to n , restrict , and redact . Then we build adversaries $\mathcal{B}_{\text{INT-PCTXT-SA}}$ and $\mathcal{B}_{\text{CORR-G-SA}}$ such that*

$$\text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CA-SA}}(\mathcal{A}_{\text{CA-SA}}) \leq \text{Adv}_{\text{GC}_{\text{SA}}, n}^{\text{INT-PCTXT-SA}}(\mathcal{B}_{\text{INT-PCTXT-SA}}) + \text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CORR-G-SA}}(\mathcal{B}_{\text{CORR-G-SA}}).$$

Proof sketch. Game $\mathcal{G}_0$ records honest sends and their delayed evidence without changing the server-aided causal-awareness game. Game $\mathcal{G}_1$ aborts if an accepted message has no matching honest send or if accepted evidence differs from the evidence produced by that send. Adversary $\mathcal{B}_{\text{INT-PCTXT-SA}}$ answers each query using the matching server-aided integrity oracle, so every bad_0 event makes $\mathcal{B}_{\text{INT-PCTXT-SA}}$ win in that game. Game $\mathcal{G}_2$ suppresses wins caused when the receiver’s redacted local graph differs from the receiver’s required view of the redacted global graph. If $\mathcal{G}_1$ does not abort, each accepted message or evidence object is associated with an honest send. Adversary $\mathcal{B}_{\text{CORR-G-SA}}$ evaluates the deterministic receive-side algorithm locally and forwards the corresponding query to its graph-correctness game. State preservation on rejection keeps the two executions synchronized until bad_1 is set, so every bad_1 event makes this adversary win. The Fundamental Lemma bounds both hops, and $\mathcal{G}_2$ has no remaining win condition.

9 Deployment costs and limitations

Worst-case communication costs. For a global causality graph $\mathcal{G} = (n, V, E)$, let $|\mathcal{G}| = |V| + |E|$ denote the number of elements in $\mathcal{G}$. Consider an execution with m sends. The global graph contains exactly m send vertices and, since each send has at most $n - 1$ corresponding receives, at most $m(n - 1)$ receive vertices and $m(n - 1)$ edges. Hence

$$|\mathcal{G}| \leq m + 2m(n - 1) = (2n - 1)m.$$

We now bound the per-send communication costs for each construction in Section 5. In $\text{GC}_{\text{naive-max}}$, every send carries the sender’s entire local graph, which in the worst case equals the global graph $\mathcal{G}$; the overhead is therefore $(2n - 1)m = O(mn)$. The GC_{max} construction includes only a graph delta with each send, omitting vertices and edges already acknowledged by every other user. GC_{max} degrades to $\text{GC}_{\text{naive-max}}$ in the worst case when no vertex or edge has been acknowledged by every user (e.g., when one user never sends), so its overhead is again $O(mn)$. In $\text{GC}_{2\text{Hop}}$, each send includes a graph delta consisting of the sender’s actions, omitting already acknowledged vertices and edges. In the same worst case as for GC_{max} , let s denote the number of messages sent by a particular sender. That sender must include its entire action history in each sent message: s send vertices for its own sends and, for each of the at most $m - s$ received messages, three elements (the receive vertex, its corresponding send vertex, and the connecting edge), for an overhead of $s + 3(m - s) = O(m)$. Chen and Fischlin observed similar costs in the two-party setting [CF24].

Worst-case computational costs. We now bound the per-send and per-recv computational costs for each construction in Section 5. We begin with the receive procedure, which is shared by all three constructions. A receive computes the union $\mathcal{G}_j^l \cup \mathcal{G}^\Delta$ of the receiver’s local graph and the received graph delta, in time $O(|\mathcal{G}_j^l| + |\mathcal{G}^\Delta|)$. Both operands are bounded by the global-graph size $(2n - 1)m$, so the worst-case per-recv computational cost is $O(mn)$.

The send procedures differ. In $\text{GC}_{\text{naive-max}}$, a send copies the sender’s entire local graph into $\mathcal{G}^\Delta$, so its cost matches the local-graph size, which is $O(mn)$ in the worst case. In GC_{max} , a sender i computes, for each of the $n - 1$ other users, the portion of its local graph $\mathcal{G}_i^l$ that user already has and subtracts it from $\mathcal{G}_i^l$. Each of these $n - 1$ iterations scans the whole local graph and subtracts the corresponding portion, at cost $O(|\mathcal{G}_i^l|) = O(mn)$, so the per-send cost is $O(mn^2)$. This complexity results from the notational simplicity of Fig. 9 rather than an inherent cost: if each user maintains the already acknowledged subgraph $\mathcal{G}_\cap$ and updates it on every receive, a send computes $\mathcal{G}^\Delta = \mathcal{G}_i^l - \mathcal{G}_\cap$ with a single subtraction, reducing the cost to $O(mn)$. In $\text{GC}_{2\text{Hop}}$ as written, a sender i computes $\text{rG2Hop}(\mathcal{G}_i^l, j)$ for each of the $n - 1$ other users, at cost $O(|\mathcal{G}_i^l|) = O(mn)$ each, so the per-send cost is $O(mn^2)$. Since $\mathcal{G}^\Delta \subseteq \mathcal{G}_i$ and $|\mathcal{G}_i| = O(m)$, a sender that maintains $\mathcal{G}_i$ and, for each other user, the latest send of i that the user has confirmed can compute $\mathcal{G}^\Delta$ with one pass over $\mathcal{G}_i$, reducing the per-send cost to $O(m)$.

Whether the worst cases discussed above occur depends on the group’s messaging pattern and the reliability of delivery. For both GC_{max} and $\text{GC}_{2\text{Hop}}$, the per-send delta contains events not yet acknowledged by every member, so its size depends on how far the least up-to-date member lags. When all members send regularly and messages are delivered reliably, every member continuously acknowledges messages, so this lag stays within a small recent window. To reduce this dependence on communication patterns, deployments can adopt the checkpoint or server-aided delivery variants.

Improvements via checkpoints and server-aided delivery. Checkpoints and server-aided delivery offer conditional savings, though neither changes the asymptotic worst case. Once applied, a checkpoint establishes a common baseline, allowing later messages to omit that shared component from their graph deltas instead of transmitting it again. However, checkpoints do not change the asymptotic worst case, which is reached when a checkpoint fails to establish a baseline shared across users (e.g., when the only user to create checkpoints never receives any message, so no other party benefits from it). An application-supplied checkpoint policy would determine when a user creates a checkpoint (e.g., when the user’s graph delta reaches x elements or once per day). Each user would evaluate this policy after each receive and before each send. Therefore, the per-message communication and computational overhead reduces to at most the threshold specified in the checkpoint policy.

Server-aided delivery similarly offers conditional savings. For senders, server-aided delivery incurs the same per-send overhead as the non-server-aided construction; a receiver’s costs, however, are lower when it does not fetch every evidence object. If a receiver fetches every evidence object, its communication cost matches the non-server-aided cost and includes the additional in-band binding.

Scope and limitations. Our model assumes static groups and honest members, while the adversary controls delivery. Extending the model to dynamic groups would require defining the global graph and the restricted graph function with respect to membership. Adding a member requires a policy that determines the new member’s baseline and how much pre-join history they may learn. Removing a member requires policies for messages sent before the removal but delivered afterward and for retaining causal references to the removed member. Moreover, group-messaging approaches such as TreeKEM [WPB25], Sender Keys [BCG23], and per-recipient transmission over pairwise Double Ratchet channels [RMS18] handle membership changes differently. Addressing dynamic groups would require treating each of these settings individually, which is beyond the scope of our work. We believe that our core technique of sending the relevant subgraph of the sender’s local graph would nonetheless apply.

Malicious members create a separate difficulty: they can lie about or omit observations, so a transmitted subgraph need not represent what they actually knew when sending. Handling malicious members remains outside our scope.

Our model also does not capture several features that deployed messengers provide. For example, message edits and deletions allow modifications to a send event after it has been delivered, whereas a vertex in our causality graphs cannot be updated once added. Deployed messengers also support multiple devices per user, each of which would act as its own sending source; our model gives each user exactly one local graph and one action counter. Capturing these features would require causality graphs whose events can supersede earlier ones and whose vertex sets are indexed by device, together with restricted graph functions that state what each device must know.

Relatedly, our model treats communication between two users as a single long-term channel. In deployment, this channel is itself realized by many pairwise sessions. Signal manages these with its Sesame layer and stores up to 40 sessions per device: one is the current active session used to encrypt, the remaining sessions are retained, and any session that can decrypt an incoming message becomes the active one [CJN23]. Messages between two users may therefore arrive over several different sessions, and a receiver must reconcile the causal information it derives from each. Our work does not model this reconciliation.

Causal awareness authenticates a selected causal view; it neither forces all users’ transcripts to converge nor imposes total-order delivery. It also does not by itself establish message confidentiality, forward secrecy, post-compromise security, or metadata privacy. Our Graph-then-Encrypt construction simply passes the graph-correct channel’s output as the plaintext payload to an encrypted channel. Therefore, those properties must come from, and be preserved by, the surrounding encrypted channel.

Acknowledgements

We thank Kyle Watters for helpful discussions on graph encoding. Akshaya Kumar was supported by an Amazon Research Award, Fall 2024.

References

- AAN⁺22. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07085-3_28. 4, 6, 49
- AAN⁺24. Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable CGKA with fast healing. In Clemente Galdi and Duong Hieu Phan, editors, *SCN 24, Part II*, volume 14974 of *LNCS*, pages 294–313. Springer, Cham, September 2024. doi:10.1007/978-3-031-71073-5_14. 4, 6, 49
- ACD19. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Cham, May 2019. doi:10.1007/978-3-030-17653-2_5. 3
- AMPS26. Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. *Journal of Cryptology*, 39(1):10, January 2026. doi:10.1007/s00145-025-09566-1. 6, 49
- AMT23. Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 396–429. Springer, Cham, August 2023. doi:10.1007/978-3-031-38551-3_13. 4, 6, 49
- ASB18. Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162–173, 2018. doi:10.1016/j.jpdc.2017.08.003. 6, 47
- BBR⁺23. R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023. URL: <https://www.rfc-editor.org/info/rfc9420/>, doi:10.17487/RFC9420. 6, 49
- BCC⁺23. Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 362–395. Springer, Cham, August 2023. doi:10.1007/978-3-031-38551-3_12. 6, 48
- BCG23. David Balbás, Daniel Collins, and Phillip Gajland. WhatsApp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023. doi:10.1007/978-981-99-8733-7_10. 26
- BR06. Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006. doi:10.1007/11761679_25. 6

- BRO⁺25. B. Beurdouche, E. Rescorla, E. Omara, S. Inguva, and A. Duric. The Messaging Layer Security (MLS) Architecture. RFC 9750, April 2025. URL: <https://www.rfc-editor.org/info/rfc9750/>, doi: [10.17487/RFC9750](https://doi.org/10.17487/RFC9750). 6, 49
- BSS91. Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991. doi: [10.1145/128738.128742](https://doi.org/10.1145/128738.128742). 6, 47
- CCD⁺20. Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020. doi: [10.1007/s00145-020-09360-1](https://doi.org/10.1007/s00145-020-09360-1). 3
- CCG⁺18. Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018. doi: [10.1145/3243734.3243747](https://doi.org/10.1145/3243734.3243747). 3
- CF24. Shan Chen and Marc Fischlin. Integrating causality in messaging channels. In Marc Joye and Gregor Leander, editors, *EUROCRYPT 2024, Part III*, volume 14653 of *LNCS*, pages 251–282. Springer, Cham, May 2024. doi: [10.1007/978-3-031-58734-4_9](https://doi.org/10.1007/978-3-031-58734-4_9). 5, 6, 14, 25, 48, 49
- CJN23. Cas Cremers, Charlie Jacomme, and Aurora Naska. Formal analysis of session-handling in secure messaging: Lifting security from sessions to conversations. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1235–1252. USENIX Association, August 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/cremers-session-handling>. 27
- CsS07. Christian Cachin, abhi shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In Indranil Gupta and Roger Wattenhofer, editors, *26th ACM PODC*, pages 129–138. ACM, August 2007. doi: [10.1145/1281100.1281121](https://doi.org/10.1145/1281100.1281121). 4, 6, 49
- DIRZ14. Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 4:1–4:13. ACM, 2014. doi: [10.1145/2670979.2670983](https://doi.org/10.1145/2670979.2670983). 6, 47
- EMP18. Patrick Eugster, Giorgia Azzurra Marson, and Bertram Poettering. A cryptographic look at multi-party channels. In Steve Chong and Stephanie Delaune, editors, *CSF 2018 Computer Security Foundations Symposium*, pages 31–45. IEEE Computer Society Press, 2018. doi: [10.1109/CSF.2018.00010](https://doi.org/10.1109/CSF.2018.00010). 5, 6, 48
- FKBS25. Konstantin Fischer, Markus Keil, Annalina Buckmann, and M. Angela Sasse. “If you want to encrypt it really, really hardcore...”: User perceptions of key transparency in WhatsApp. *PoPETs*, 2025(4):1039–1054, October 2025. doi: [10.56553/popets-2025-0170](https://doi.org/10.56553/popets-2025-0170). 4
- GGG⁺26. Gabriel K. Gegenhuber, Moritz Grefner, Maximilian Günther, Matthäus Wining, David Schmidt, and Aljosha Judmayer. Send and Pretend: Exploiting transcript consistency issues in end-to-end encrypted group chats. In *USENIX Security 2026*, pages 7115–7134. USENIX Association, August 2026. URL: <https://www.usenix.org/conference/usenixsecurity26/presentation/gegenhuber>. 6, 49
- GUVC09. Ian Goldberg, Berkant Ustaoglu, Matthew Van Gundy, and Hao Chen. Multi-party off-the-record messaging. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 2009*, pages 358–368. ACM Press, November 2009. doi: [10.1145/1653662.1653705](https://doi.org/10.1145/1653662.1653705). 6, 48
- JBHH21. Florian Jacob, Carolin Beer, Norbert Henze, and Hannes Hartenstein. Analysis of the Matrix Event Graph replicated data type. *IEEE Access*, 9:28317–28333, 2021. doi: [10.1109/ACCESS.2021.3058576](https://doi.org/10.1109/ACCESS.2021.3058576). 6, 48
- JK25. Joseph Jaeger and Akshaya Kumar. Analyzing group chat encryption in MLS, session, signal, and matrix. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VIII*, volume 15608 of *LNCS*, pages 272–301. Springer, Cham, May 2025. doi: [10.1007/978-3-031-91101-9_10](https://doi.org/10.1007/978-3-031-91101-9_10). 3, 6, 49
- Lam78. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. doi: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563). 5, 6, 8, 47
- LFKA11. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don’t settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011. doi: [10.1145/2043556.2043593](https://doi.org/10.1145/2043556.2043593). 6, 47
- LZPR24. Chen-Da Liu-Zhang, Christopher Portmann, and Guilherme Rito. Stateful communication with malicious parties. Cryptology ePrint Archive, Report 2024/1593, 2024. URL: <https://eprint.iacr.org/2024/1593>. 6, 48
- Mar14. Moxie Marlinspike. Private group messaging. Open Whisper Systems blog, 2014. <https://signal.org/blog/private-groups/>. 6, 48

- Mar17. Giorgia Azzurra Marson. *Real-World Aspects of Secure Channels: Fragmentation, Causality, and Forward Security*. PhD thesis, Technische Universität Darmstadt, 2017. doi:10.26083/tuprints-00006021. 5, 48
- NMM18. Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. Causal broadcast: How to forget? In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics*, pages 20:1–20:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.OPODIS.2018.20. 6
- NR25. Armin Namavari and Thomas Ristenpart. Transcript franking for encrypted messaging. In Goichiro Hanaoka and Bo-Yin Yang, editors, *ASIACRYPT 2025, Part II*, volume 16246 of *LNCS*, pages 3–33. Springer, Singapore, December 2025. doi:10.1007/978-981-95-5096-8_1. 6, 48, 49
- PST23. Kenneth G. Paterson, Matteo Scarlata, and Kien T. Truong. Three lessons from Threema: Analysis of a secure messenger. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1289–1306. USENIX Association, August 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/paterson>. 6, 49
- RB94. Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM TOPLAS*, 16(3):986–1009, 1994. 5, 6, 48
- RG93. Michael K. Reiter and Li Gong. Preventing denial and forgery of causal relationships in distributed systems. In *1993 IEEE Symposium on Security and Privacy*, pages 30–40. IEEE Computer Society Press, 1993. doi:10.1109/RISP.1993.287645. 5, 6, 48
- RMS18. Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy*, pages 415–429. IEEE Computer Society Press, April 2018. doi:10.1109/EuroSP.2018.00036. 3, 5, 6, 26, 49
- SH19. Michael Schliep and Nicholas Hopper. End-to-end secure mobile group messaging with conversation integrity and deniability. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 55–73. ACM, November 2019. doi:10.1145/3338498.3358644. 6, 48
- SKH17. Michael Schliep, Ian Kariniemi, and Nicholas Hopper. Is Bob sending mixed signals? In *Proceedings of the 2017 Workshop on Privacy in the Electronic Society (WPES)*, pages 31–40. ACM, October 2017. doi:10.1145/3139550.3139568. 6, 49
- SVH18. Michael Schliep, Eugene Y. Vasserman, and Nicholas Hopper. Consistent synchronous group off-the-record messaging with SYM-GOTR. *PoPETs*, 2018(3):181–202, July 2018. doi:10.1515/popets-2018-0027. 6, 48
- UDB⁺15. Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE Computer Society Press, May 2015. doi:10.1109/SP.2015.22. 3, 5, 6, 48
- WKHB21. Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, November 2021. doi:10.1145/3460120.3484542. 4, 6, 49
- WPB25. Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. TreeKEM: A modular machine-checked symbolic security analysis of group key agreement in messaging layer security. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 4375–4390. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00228. 5, 26
- WPBB23. Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1217–1233. USENIX Association, August 2023. URL: <https://www.usenix.org/conference/usenixsecurity23/presentation/wallez>. 5
- WT99. Alma Whitten and J. Doug Tygar. Why johnny can’t encrypt: A usability evaluation of PGP 5.0. In G. Winfield Treese, editor, *USENIX Security 99*. USENIX Association, August 1999. URL: <https://www.usenix.org/conference/8th-usenix-security-symposium/why-johnny-cant-encrypt-usability-evaluation-pgp-50>. 4

Game $\mathcal{G}_{\text{GC}_{\text{SA}},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}\text{-SA}}(\mathcal{A}_{\text{CORR-}\mathcal{G}\text{-SA}})$ win $\leftarrow$ false ; $\mathcal{G} \leftarrow \text{InitGraph}(n)$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$ $\mathcal{A}_{\text{CORR-}\mathcal{G}\text{-SA}}^{\text{SEND,RECV,VERIFY}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$; Return win <u>VERIFY(i, j, ctr)</u> // $i, j \in [n], \text{ctr} \in \mathbb{N}$ require pendingToken[i, j, ctr] $\neq \perp$ verTok $\leftarrow$ pendingToken[i, j, ctr] evidence $\leftarrow$ sendEvidence[i, ctr] $(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence})$ If acc = true then pendingToken[i, j, ctr] $\leftarrow \perp$ $\mathcal{G} \leftarrow \text{AddEvidenceReceipt}(\mathcal{G}, i, j, \text{ctr})$ If redact($\mathcal{G}_j^l$) $\neq$ restrict(redact($\mathcal{G}$), j) then win $\leftarrow$ true Return ($\mathcal{G}_j^l, \text{acc}$)	<u>SEND(i, m)</u> // $i \in [n], m \in \text{GC}_{\text{SA}}.\text{MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ sendEvidence[i, ctr_i] $\leftarrow$ evidence $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ $C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$ Return ($st_i, \mathcal{G}_i^l, c, \text{evidence}$) <u>RECV($i, j, \text{ctr}$)</u> // $i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$ require $C[i, j, \text{ctr}] \neq \perp$ $c \leftarrow C[i, j, \text{ctr}]$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}', \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ pendingToken[i, j, ctr] $\leftarrow$ verTok $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$ If redact($\mathcal{G}_j^l$) $\neq$ restrict(redact($\mathcal{G}$), j) then win $\leftarrow$ true $C[i, j, \text{ctr}] \leftarrow \perp$; Return ($\mathcal{G}_j^l, m, \text{ctr}'$)
Game $\mathcal{G}_{\text{GC}_{\text{SA}},n}^{\text{INT-}\mathcal{x}\text{-SA}}(\mathcal{A}_{\text{INT-}\mathcal{x}\text{-SA}})$ win $\leftarrow$ false ; $S \leftarrow \emptyset$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$ $\mathcal{A}_{\text{INT-}\mathcal{x}\text{-SA}}^{\text{SEND,RECV,VERIFY}}$; Return win <u>VERIFY($i, j, \text{ctr}, \text{evidence}'$)</u> // $i, j \in [n], \text{ctr} \in \mathbb{N}, \text{evidence}' \in \{0, 1\}^*$ require pendingToken[i, j, ctr] $\neq \perp$ verTok $\leftarrow$ pendingToken[i, j, ctr] $(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence}')$ If acc = true then If evidence' $\neq$ sendEvidence[i, ctr] then win $\leftarrow$ true pendingToken[i, j, ctr] $\leftarrow \perp$ Return ($\mathcal{G}_j^l, \text{acc}$)	<u>SEND(i, m)</u> // $i \in [n], m \in \text{GC}_{\text{SA}}.\text{MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ sendEvidence[i, ctr_i] $\leftarrow$ evidence $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$ // INT-PCTXT $S \leftarrow S \cup \{(i, j, c) \mid j \in [n] \setminus \{i\}\}$ // INT-CTXT $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m) \mid j \in [n] \setminus \{i\}\}$ // INT-PTXT Return ($\mathcal{G}_i^l, c, \text{evidence}$) <u>RECV($i, j, c$)</u> // $i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}, \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ entry $\leftarrow (i, j, \text{ctr}, m, c)$ // INT-PCTXT entry $\leftarrow (i, j, c)$ // INT-CTXT entry $\leftarrow (i, j, \text{ctr}, m)$ // INT-PTXT If entry $\notin S$ then win $\leftarrow$ true Else $S \leftarrow S \setminus \{\text{entry}\}$ pendingToken[i, j, ctr] $\leftarrow$ verTok Return ($\mathcal{G}_j^l, m, \text{ctr}$)
Game $\mathcal{G}_{\text{GC}_{\text{SA}},n,\text{restrict},\text{redact}}^{\text{CA-SA}}(\mathcal{A}_{\text{CA-SA}})$ win $\leftarrow$ false ; $\mathcal{G} \leftarrow \text{InitGraph}(n)$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$ $\mathcal{A}_{\text{CA-SA}}^{\text{SEND,RECV,VERIFY}}$; Return win <u>RECV(i, j, c)</u> // $i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}, \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ pendingToken[i, j, ctr] $\leftarrow$ verTok $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$ If redact($\mathcal{G}_j^l$) $\neq$ restrict(redact($\mathcal{G}$), j) then win $\leftarrow$ true Return ($\mathcal{G}_j^l, m, \text{ctr}$)	<u>SEND(i, m)</u> // $i \in [n], m \in \text{GC}_{\text{SA}}.\text{MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ Return ($\mathcal{G}_i^l, c, \text{evidence}$) <u>VERIFY($i, j, \text{ctr}, \text{evidence}'$)</u> // $i, j \in [n], \text{ctr} \in \mathbb{N}, \text{evidence}' \in \{0, 1\}^*$ require pendingToken[i, j, ctr] $\neq \perp$ verTok $\leftarrow$ pendingToken[i, j, ctr] $(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence}')$ If acc = false then return ($\mathcal{G}_j^l, \text{false}$) pendingToken[$i, j, \text{ctr}$] $\leftarrow \perp$ $\mathcal{G} \leftarrow \text{AddEvidenceReceipt}(\mathcal{G}, i, j, \text{ctr})$ If redact($\mathcal{G}_j^l$) $\neq$ restrict(redact($\mathcal{G}$), j) then win $\leftarrow$ true Return ($\mathcal{G}_j^l, \text{acc}$)

Fig. 21: **Top pane:** Game $\mathcal{G}^{\text{CORR-}\mathcal{G}\text{-SA}}$ defining graph correctness of GC_{SA} with parameters n , restrict, and redact. **Middle pane:** Game $\mathcal{G}^{\text{INT-}\mathcal{x}\text{-SA}}$ defining integrity of GC_{SA} with parameters n and $x \in \{\text{PCTXT}, \text{CTXT}, \text{PTXT}\}$. **Bottom pane:** Game $\mathcal{G}^{\text{CA-SA}}$ defining causal awareness of GC_{SA} with parameters n , restrict, and redact.

Game $\mathcal{G}$	$\text{RECV}(i, j, \text{ctr}) \quad // \quad i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$
$\text{win} \leftarrow \text{false}$	require $C[i, j, \text{ctr}] \neq \perp$
$\mathcal{G} \leftarrow \text{InitGraph}(n)$	$p \leftarrow C[i, j, \text{ctr}]$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$(\mathcal{G}^\Delta, m, \text{ctr}') \leftarrow p$
$st_i \leftarrow 0, \mathcal{G}_i^l \leftarrow \text{InitGraph}(n) \quad \forall i \in [n]$	$\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$
$\mathcal{A}_{\text{CORR-}\mathcal{G}}^{\text{SEND, RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	$\mathcal{G}_j^l \leftarrow \text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr}')$
Return win	If $m = \perp$ then return $\perp$
$\text{SEND}(i, m) \quad // \quad i \in [n], m \in \text{GC}_{\text{naive-max}}.\text{MS}$	$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$
$\text{ctr}_i \leftarrow \text{ctr}_i + 1$	If $\text{redactLbl}(\mathcal{G}_j^l) \neq \text{rGMax}(\text{redactLbl}(\mathcal{G}), j)$ then
$st_i \leftarrow st_i + 1$	$\text{win} \leftarrow \text{true}$
$\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$	$C[i, j, \text{ctr}] \leftarrow \perp$
$\mathcal{G}^\Delta \leftarrow \mathcal{G}_i^l$	Return $(\mathcal{G}_j^l, m, \text{ctr}')$
$p \leftarrow (\mathcal{G}^\Delta, m, st_i)$	
$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, p))$	
$C[i, j, \text{ctr}_i] \leftarrow p \quad \forall j \in [n] \setminus \{i\}$	
Return $(st_i, \mathcal{G}_i^l, p)$	

Fig. 22: Game $\mathcal{G}$ used in the proof of [Theorem 1](#). The lines in gray were added by expanding $\text{GC}_{\text{naive-max}}.\text{Init}$, $\text{GC}_{\text{naive-max}}.\text{Send}$, and $\text{GC}_{\text{naive-max}}.\text{Recv}$ in the game $\mathcal{G}_{\text{GC}_{\text{naive-max}}, \mathcal{M}, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}$.

A Proofs for group chat constructions

A.1 Proof of [Theorem 1](#)

The game $\mathcal{G}$ in [Fig. 22](#) is obtained by expanding multiple lines in $\mathcal{G}_{\text{GC}_{\text{naive-max}}, \mathcal{M}, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}$, where the expanded lines are marked in gray. To prove [Theorem 1](#), we establish the following invariant throughout this game:

$$\text{redactLbl}(\mathcal{G}_i^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), i) \quad \forall i \in [n].$$

If this invariant holds, the win flag is never set in RECV and $\Pr[\mathcal{G}_{\text{GC}_{\text{naive-max}}, \mathcal{M}, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}})] = 0$.

Initialization. Both $\mathcal{G}_i^l$ and $\mathcal{G}$ are initialized to the empty graph, so $\text{rGMax}(\text{redactLbl}(\mathcal{G}), i) = \text{redactLbl}(\mathcal{G}_i^l)$ holds trivially for all $i \in [n]$.

Send. Consider any $\text{SEND}(i, m)$ call. Let $\mathcal{G}_i^l$ and $\mathcal{G}$ be i 's local graph and the global graph respectively at the beginning of the current SEND call, and $\mathcal{G}_i^{l'}$ and $\mathcal{G}'$ be the same graphs at the end of the SEND call. Assuming the invariant at the beginning of the send call, we have $\text{rGMax}(\text{redactLbl}(\mathcal{G}), i) = \text{redactLbl}(\mathcal{G}_i^l)$. We show that $\text{rGMax}(\text{redactLbl}(\mathcal{G}'), i) = \text{redactLbl}(\mathcal{G}_i^{l'})$.

In the current SEND call, the $\text{GC}_{\text{naive-max}}.\text{Send}$ adds a new sending vertex u to $\mathcal{G}_i^l$ with send-event identifier st_i and label $\perp$ to obtain $\mathcal{G}_i^{l'}$; the game adds the same vertex to $\mathcal{G}$ with label (m, p) to obtain $\mathcal{G}'$. Since redactLbl removes the graph label, the vertices added to $\mathcal{G}_i^l$ and $\mathcal{G}$ are identical after redaction. The newly added vertices now become v_{max}^i in $\mathcal{G}_i^{l'}$ and $\mathcal{G}'$. After redaction, $\text{rGMax}(\text{redactLbl}(\mathcal{G}'), i)$ and $\text{redactLbl}(\mathcal{G}_i^{l'})$ extend the two sides of the invariant by the same vertex. Hence $\text{rGMax}(\text{redactLbl}(\mathcal{G}'), i) = \text{redactLbl}(\mathcal{G}_i^{l'})$. No edges or vertices were added for any $j \in [n] \setminus \{i\}$, so $\text{rGMax}(\text{redactLbl}(\mathcal{G}), j) = \text{redactLbl}(\mathcal{G}_j^l)$ continues to hold.

Receive. Consider any $\text{RECV}(i, j, \text{ctr})$ call. Let $\mathcal{G}_j^l$ and $\mathcal{G}$ be j 's local graph and the global graph respectively at the beginning of the current RECV call, and $\mathcal{G}_j^{l'}$ and $\mathcal{G}'$ be the same graphs at the end of the call. Assuming the invariant at the beginning of the call, we have $\text{rGMax}(\text{redactLbl}(\mathcal{G}), j) = \text{redactLbl}(\mathcal{G}_j^l)$. We show that $\text{rGMax}(\text{redactLbl}(\mathcal{G}'), j) = \text{redactLbl}(\mathcal{G}_j^{l'})$.

Let u be i 's sending vertex with send-event identifier ctr and v be the new receive vertex of j . The payload stores $\mathcal{G}^\Delta = \mathcal{G}_i^l$ as captured during the matching $\text{SEND}(i, m)$ call. By the invariant at the end of that send call, $\text{redactLbl}(\mathcal{G}^\Delta)$ equals $\text{rGMax}(\text{redactLbl}(\mathcal{G}_{\text{at send}}), i)$, i.e., $\mathcal{G}^\Delta$ contains all vertices $w \in \mathcal{G}_{\text{at send}}$ such that $w \leq u$. Since the causal ancestors of any fixed vertex u are unchanged by later additions, $\mathcal{G}^\Delta$

Game $\mathcal{G}$	$\text{SEND}(i, m) \quad // \ i \in [n], m \in \{0, 1\}^*$
$\text{win} \leftarrow \text{false}$	$\text{ctr}_i \leftarrow \text{ctr}_i + 1$
$\mathcal{G} \leftarrow \text{InitGraph}(n)$	$st_i \leftarrow st_i + 1$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$
$st_i \leftarrow 0, \mathcal{G}_i^l \leftarrow \text{InitGraph}(n) \quad \forall i \in [n]$	$(n, V, E) \leftarrow \mathcal{G}_i^l$
$\mathcal{A}_{\text{CORR-}\mathcal{G}}^{\text{SEND, RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	For all $j \in [n] \setminus \{i\}$ do
Return win	$v_{\max}^j \leftarrow \max(V_j, <_\ell)$
$\text{RECV}(i, j, \text{ctr}) \quad // \ i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$	$V_j' \leftarrow \{v \in V : v \leq v_{\max}^j\}$
require $C[i, j, \text{ctr}] \neq \perp$	$E_j' \leftarrow \{(x, y) \in E : x, y \in V_j'\}$
$p \leftarrow C[i, j, \text{ctr}]$	$\mathcal{G}_j \leftarrow (n, V_j', E_j')$
$(\mathcal{G}^\Delta, m, \text{ctr}') \leftarrow p$	$\mathcal{G}^\Delta \leftarrow \bigcup_{j \in [n] \setminus \{i\}} (\mathcal{G}_i^l - \mathcal{G}_j)$
$\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$	$p \leftarrow (\mathcal{G}^\Delta, m, st_i)$
$\mathcal{G}_j^l \leftarrow \text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr}')$	$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, p))$
If $m = \perp$ then return $\perp$	$C[i, j, \text{ctr}_i] \leftarrow p \quad \forall j \in [n] \setminus \{i\}$
$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$	Return $(st_i, \mathcal{G}_i^l, p)$
If $\text{redactLbl}(\mathcal{G}_j^l) \neq \text{rGMax}(\text{redactLbl}(\mathcal{G}), j)$ then	
$\text{win} \leftarrow \text{true}$	
$C[i, j, \text{ctr}] \leftarrow \perp$	
Return $(\mathcal{G}_j^l, m, \text{ctr}')$	

Fig. 23: Game $\mathcal{G}$ used in the proof of [Theorem 2](#). Lines in gray expand $\text{GC}_{\max}.\text{Init}$, $\text{GC}_{\max}.\text{Send}$, and $\text{GC}_{\max}.\text{Recv}$ in $\mathcal{G}_{\text{GC}_{\max}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}$.

still contains all the causal ancestors of u in the current $\mathcal{G}$. By the assumption at the beginning of the current RECV call, $\text{redactLbl}(\mathcal{G}_j^l)$ equals $\text{rGMax}(\text{redactLbl}(\mathcal{G}), j)$, i.e., $\mathcal{G}_j^l$ contains the causal ancestors of j 's previous latest event v_{prev}^j . After $\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$ and $\text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr})$, the receive vertex v satisfies $v_{\text{prev}}^j \leq v$ and $(u, v) \in \mathcal{G}'$. The vertices causally preceding v in $\mathcal{G}'$ are exactly the vertices preceding u , the vertices preceding v_{prev}^j , and the new vertex v itself. The corresponding edge set is the union of the edges already induced by those two predecessor sets together with the new delivery edge (u, v) . These are exactly the vertices and edges in $\mathcal{G}_j^{l'}$ after redaction, so $\text{redactLbl}(\mathcal{G}_j^{l'}) = \text{rGMax}(\text{redactLbl}(\mathcal{G}'), j)$.

Since the invariant holds after every oracle call, the check in RECV is always false, win is never set, and $\text{Adv}_{\text{GC}_{\text{naive-max}}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}}) = 0$.

A.2 Proof of [Theorem 2](#)

The game $\mathcal{G}$ in [Fig. 23](#) is obtained by expanding multiple lines in $\mathcal{G}_{\text{GC}_{\max}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}$, where the expanded lines are marked in gray. To prove [Theorem 2](#), we establish the following invariant throughout this game:

$$\text{redactLbl}(\mathcal{G}_i^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), i) \quad \forall i \in [n].$$

If this invariant holds, the win flag is never set in RECV and $\Pr[\mathcal{G}_{\text{GC}_{\max}, n, \text{rGMax}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}})] = 0$.

Initialization, Send. These cases follow identically to the same cases in the proof of [Theorem 1](#).

Receive. The $\text{GC}_{\max}.\text{Recv}$ algorithm is identical to $\text{GC}_{\text{naive-max}}.\text{Recv}$: both schemes execute $\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$ followed by AddRecvVrtx ; we established the invariant for $\text{GC}_{\text{naive-max}}$ by substituting $\mathcal{G}^\Delta = \mathcal{G}_i^l$. To reuse the argument we used in the proof of [Theorem 1](#), it suffices to show that $\text{GC}_{\max}$'s $\mathcal{G}^\Delta = \bigcup_{k \in [n] \setminus \{i\}} (\mathcal{G}_i^l - \mathcal{G}_k)$ yields the same updated graph after redaction, i.e., to show that

$$\text{redactLbl}\left(\mathcal{G}_j^l \cup \bigcup_{k \in [n] \setminus \{i\}} (\mathcal{G}_i^l - \mathcal{G}_k)\right) = \text{redactLbl}(\mathcal{G}_j^l \cup \mathcal{G}_i^l).$$

We can rewrite the graph inside the redaction on the left as

Game $\mathcal{G}$	$\text{SEND}(i, m) \text{ // } i \in [n], m \in \{0, 1\}^*$
$\text{win} \leftarrow \text{false}$	$\text{ctr}_i \leftarrow \text{ctr}_i + 1$
$\mathcal{G} \leftarrow \text{InitGraph}(n)$	$st_i \leftarrow st_i + 1$
$\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$	$\mathcal{G}_i^l \leftarrow \text{AddSendVrtx}(\mathcal{G}_i^l, i, st_i, \perp)$
$st_i \leftarrow 0, \mathcal{G}_i^l \leftarrow \text{InitGraph}(n) \quad \forall i \in [n]$	$(n, V, E) \leftarrow \mathcal{G}_i^l$
$\mathcal{A}_{\text{CORR-}\mathcal{G}}^{\text{SEND, RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	$\mathcal{G}_i \leftarrow (n, V_i, \{\})$
Return win	$\cup (n, \{\}, \{(x, y) \in E : y \in V_i\})$
$\text{RCV}(i, j, \text{ctr}) \text{ // } i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$	$\cup (n, \{x : (x, y) \in E \wedge y \in V_i\}, \{\})$
require $C[i, j, \text{ctr}] \neq \perp$	For all $j \in [n] \setminus \{i\}$ do
$p \leftarrow C[i, j, \text{ctr}]$	$\mathcal{G}_j \leftarrow \text{rG2Hop}(\mathcal{G}_i^l, j)$
$(\mathcal{G}^\Delta, m, \text{ctr}') \leftarrow p$	$\mathcal{G}^\Delta \leftarrow \bigcup_{j \in [n] \setminus \{i\}} (\mathcal{G}_i - \mathcal{G}_j)$
$\mathcal{G}_j^l \leftarrow \mathcal{G}_j^l \cup \mathcal{G}^\Delta$	$\text{ctr} \leftarrow st_i ; p \leftarrow (\mathcal{G}^\Delta, m, \text{ctr}')$
$\mathcal{G}_j^l \leftarrow \text{AddRecvVrtx}(\mathcal{G}_j^l, i, j, \text{ctr}')$	$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, p))$
If $m = \perp$ then return $\perp$	$C[i, j, \text{ctr}_i] \leftarrow p \quad \forall j \in [n] \setminus \{i\}$
$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$	Return $(st_i, \mathcal{G}_i^l, p)$
If $\text{redactLbl}(\mathcal{G}_j^l) \neq \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j)$ then	
win $\leftarrow \text{true}$	
$C[i, j, \text{ctr}] \leftarrow \perp$	
Return $(\mathcal{G}_j^l, m, \text{ctr}')$	

Fig. 24: Game $\mathcal{G}$ used in the proof of [Theorem 4](#). Lines in gray expand $\text{GC}_{2\text{Hop}}.\text{Init}$, $\text{GC}_{2\text{Hop}}.\text{Send}$, and $\text{GC}_{2\text{Hop}}.\text{Recv}$ in $\mathcal{G}_{\text{GC}_{2\text{Hop}}, n, \text{rG2Hop}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}$.

$$\begin{aligned}
& \mathcal{G}_j^l \cup \bigcup_{k \in [n] \setminus \{i\}} (\mathcal{G}_i^l - \mathcal{G}_k) \\
&= \mathcal{G}_j^l \cup (\mathcal{G}_i^l - (\bigcap_{k \in [n] \setminus \{i\}} \mathcal{G}_k)) \\
&= (\mathcal{G}_j^l \cup \mathcal{G}_i^l) - (\bigcap_{k \in [n] \setminus \{i\}} \mathcal{G}_k - \mathcal{G}_j^l).
\end{aligned}$$

Now we show that $(\bigcap_{k \in [n] \setminus \{i\}} \mathcal{G}_k - \mathcal{G}_j^l)$ is empty after redaction. Since $j \in [n] \setminus \{i\}$, the receiver j is one of the parties indexed in the intersection, so

$$\bigcap_{k \in [n] \setminus \{i\}} \mathcal{G}_k \subseteq \mathcal{G}_j.$$

It therefore suffices to show $\text{redactLbl}(\mathcal{G}_j) \subseteq \text{redactLbl}(\mathcal{G}_j^l)$. By the invariant at the matching SEND call, $\text{redactLbl}(\mathcal{G}_i^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}_{\text{at send}}), i)$, which gives $\text{redactLbl}(\mathcal{G}_i^l) \subseteq \text{redactLbl}(\mathcal{G}_{\text{at send}})$. Since $\text{redactLbl}(\mathcal{G}_j) = \text{rGMax}(\text{redactLbl}(\mathcal{G}_i^l), j)$ and $\text{redactLbl}(\mathcal{G}_i^l) \subseteq \text{redactLbl}(\mathcal{G}_{\text{at send}}) \subseteq \text{redactLbl}(\mathcal{G})$ (the global graph is monotone), we have

$$\text{redactLbl}(\mathcal{G}_j) \subseteq \text{rGMax}(\text{redactLbl}(\mathcal{G}), j).$$

By the invariant at the beginning of the current RCV call, $\text{redactLbl}(\mathcal{G}_j^l) = \text{rGMax}(\text{redactLbl}(\mathcal{G}), j)$, so $\text{redactLbl}(\mathcal{G}_j) \subseteq \text{redactLbl}(\mathcal{G}_j^l)$. Combining these inclusions after redaction shows that $\bigcap_{k \in [n] \setminus \{i\}} \mathcal{G}_k \setminus \mathcal{G}_j^l$ is empty after redaction.

A.3 Proof of [Theorem 4](#)

We prove [Theorem 4](#) by showing that the invariant

$$\text{redactLbl}(\mathcal{G}_i^l) = \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), i) \quad \forall i \in [n]$$

holds throughout the game in [Fig. 24](#). Therefore the check in RCV is always false, the win flag is never set, and

$$\Pr[\mathcal{G}_{\text{GC}_{2\text{Hop}}, n, \text{rG2Hop}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}})] = 0.$$

Initialization and send. Initialization is immediate because all local graphs and the global graph are empty. A send by i adds the same new i -vertex to $\mathcal{G}_i^l$ and to the global graph, up to labels erased by `redactLbl`, and does not change the restricted graph of any other user. Thus the invariant is preserved by every `SEND` call.

Receive. Consider a receive query `RECV`(i, j, ctr). Let $\mathcal{G}$ and $\mathcal{G}'$ be the global graph immediately before and after the `RECV` call respectively. Let $\mathcal{G}_j^l$ and $\mathcal{G}_j^{l'}$ be j 's local graph immediately before and after the `RECV` call respectively. Let v be the receiving vertex added to the global graph in this `RECV` execution and u be the matching sending vertex of i . The global graph update during the `RECV` call gives $\mathcal{G}' = \mathcal{G} \cup (n, \{v\}, \{(u, v)\})$ (note that $u \in \mathcal{G}$ since the corresponding `SEND` query happened before this `RECV` query).

Assuming $\text{redactLbl}(\mathcal{G}_j^l) = \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j)$, we need to show that

$$\text{redactLbl}(\mathcal{G}_j^{l'}) = \text{rG2Hop}(\text{redactLbl}(\mathcal{G}'), j)$$

This is equivalent to showing that

$$\text{redactLbl}(\mathcal{G}_j^{l'}) - \text{redactLbl}(\mathcal{G}_j^l) = \text{rG2Hop}(\text{redactLbl}(\mathcal{G}'), j) - \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j). \quad (4)$$

We first rewrite the right hand side of Equation (4). Note that

$$\begin{aligned} \text{rG2Hop}(\text{redactLbl}(\mathcal{G}'), j) &= \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j) \cup (n, \{v\}, \{(u, v)\}) \\ &\quad \cup (n, V_i \cup \{w : (w, x) \in E \wedge x \in V_i\}, \{(w, x) \in E : x \in V_i\}). \end{aligned}$$

By the construction of $\mathcal{G}_i$ in the corresponding `SEND` call, we have

$$(n, V_i \cup \{w : (w, x) \in E \wedge x \in V_i\}, \{(w, x) \in E : x \in V_i\}) = \mathcal{G}_i,$$

where $(n, V, E) = \mathcal{G}_i^l$ is the local graph of i after the `SEND` call and V_i is the set of all i -vertices in V . Therefore,

$$\text{rG2Hop}(\text{redactLbl}(\mathcal{G}'), j) - \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j) = (n, \{v\}, \{(u, v)\}) \cup \mathcal{G}_i - \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j). \quad (5)$$

Now consider the left hand side of Equation (4). By the definition of `GC2Hop.Recv`, we have

$$\text{redactLbl}(\mathcal{G}_j^{l'}) = \text{redactLbl}(\mathcal{G}_j^l) \cup (n, \{v\}, \{(u, v)\}) \cup \mathcal{G}^\Delta$$

where $\mathcal{G}^\Delta$ is defined as

$$\mathcal{G}^\Delta = \bigcup_{k \in [n] \setminus \{i\}} (\mathcal{G}_i - \text{rG2Hop}(\mathcal{G}_i^l, k)) = \mathcal{G}_i - \bigcap_{k \in [n] \setminus \{i\}} \text{rG2Hop}(\mathcal{G}_i^l, k).$$

Let $\mathcal{G}^\cap = \bigcap_{k \in [n] \setminus \{i\}} \text{rG2Hop}(\mathcal{G}_i^l, k)$, so that $\mathcal{G}^\Delta = \mathcal{G}_i - \mathcal{G}^\cap$. Since $j \in [n] \setminus \{i\}$,

$$\mathcal{G}^\cap \subseteq \text{rG2Hop}(\mathcal{G}_i^l, j).$$

Moreover, since $\mathcal{G}_i^l \subseteq \text{redactLbl}(\mathcal{G})$,

$$\mathcal{G}^\cap \subseteq \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j).$$

Finally, since $\text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j) = \text{redactLbl}(\mathcal{G}_j^l)$ (this follows from our initial assumption),

$$\mathcal{G}^\cap \subseteq \text{redactLbl}(\mathcal{G}_j^l).$$

Now consider $\mathcal{G}^\Delta - \text{redactLbl}(\mathcal{G}_j^l)$. We have

$$\mathcal{G}^\Delta - \text{redactLbl}(\mathcal{G}_j^l) = (\mathcal{G}_i - \mathcal{G}^\cap) - \text{redactLbl}(\mathcal{G}_j^l) = \mathcal{G}_i - (\mathcal{G}^\cap \cup \text{redactLbl}(\mathcal{G}_j^l)).$$

Games $\mathcal{G}_0$–$\mathcal{G}_2$ $\text{win} \leftarrow \text{false}; \mathcal{G} \leftarrow \text{InitGraph}(n)$ $(\text{bad}_0, \text{bad}_1, S) \leftarrow (\text{false}, \text{false}, \emptyset)$ $\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$ $(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC.Init}(n)$ $\mathcal{A}_{\text{CA}}^{\text{SEND, RECV}}; \text{Return win}$ $\text{SEND}(i, m) \quad // i \in [n], m \in \text{GC.MS}$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{GC.Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$ $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$ $\text{Return } (\mathcal{G}_i^l, c)$ $\text{RCV}(i, j, c) \quad // i, j \in [n], i \neq j, c \in \{0, 1\}^*$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}) \leftarrow \text{GC.Rcv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ $\text{entry} \leftarrow (i, j, \text{ctr}, m, c)$ If $\text{entry} \notin S$ then $\text{bad}_0 \leftarrow \text{true}$ $\text{abort}(\text{false}) \quad // \mathcal{G}_{[1, \infty)}$ Else $S \leftarrow S \setminus \{\text{entry}\}$ $\mathcal{G} \leftarrow \text{AddRcvVrtx}(\mathcal{G}, i, j, \text{ctr})$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then $\text{bad}_1 \leftarrow \text{true}$ $\text{win} \leftarrow \text{true} \quad // \mathcal{G}_{[0, 2)}$ $\text{Return } (\mathcal{G}_j^l, m, \text{ctr})$	$\mathcal{B}_{\text{INT-PCTXT}}^{\text{SEND, RECV}}$ $\mathcal{A}_{\text{CA}}^{\text{SIMSEND, SIMRCV}}; \text{Return } \perp$ $\text{SIMSEND}(i, m)$ $\text{Return SEND}(i, m)$ $\text{SIMRCV}(i, j, c)$ $\text{Return RCV}(i, j, c)$ $\mathcal{B}_{\text{CORR-G}}^{\text{SEND, RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ $S \leftarrow \emptyset$ $\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$ $\mathcal{A}_{\text{CA}}^{\text{SIMSEND, SIMRCV}}$ $\text{SIMSEND}(i, m)$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i, \mathcal{G}_i^l, c) \leftarrow \text{SEND}(i, m)$ $S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$ $\text{Return } (\mathcal{G}_i^l, c)$ $\text{SIMRCV}(i, j, c)$ $(st_j, \mathcal{G}_j^l, m, \text{ctr}) \leftarrow \text{GC.Rcv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ $\text{entry} \leftarrow (i, j, \text{ctr}, m, c)$ If $\text{entry} \notin S$ then $\text{abort}(\perp)$ $S \leftarrow S \setminus \{\text{entry}\}$ $\text{RCV}(i, j, \text{ctr})$ $\text{Return } (\mathcal{G}_j^l, m, \text{ctr})$
---	--

Fig. 25: **Left pane:** Games $\mathcal{G}_0$ – $\mathcal{G}_2$ for the proof of [Theorem 5](#). **Right pane:** Corresponding adversaries $\mathcal{B}_{\text{INT-PCTXT}}$ (top) and $\mathcal{B}_{\text{CORR-G}}$ (bottom). Code added for game transitions is highlighted in **green**; the adversaries' calls to the oracles of their respective security games are highlighted in **orange**.

Using $\mathcal{G}^\cap \subseteq \text{redactLbl}(\mathcal{G}_j^l)$ in the above gives

$$\mathcal{G}^\Delta - \text{redactLbl}(\mathcal{G}_j^l) = \mathcal{G}_i - \text{redactLbl}(\mathcal{G}_j^l) = \mathcal{G}_i - \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j).$$

We now rewrite the left hand side of [Equation \(4\)](#) as

$$\begin{aligned} \text{redactLbl}(\mathcal{G}_j^{l'}) - \text{redactLbl}(\mathcal{G}_j^l) &= ((n, \{v\}, \{(u, v)\}) \cup \mathcal{G}^\Delta) - \text{redactLbl}(\mathcal{G}_j^l) \\ &= (n, \{v\}, \{(u, v)\}) \cup (\mathcal{G}^\Delta - \text{redactLbl}(\mathcal{G}_j^l)) \\ &= (n, \{v\}, \{(u, v)\}) \cup (\mathcal{G}_i - \text{rG2Hop}(\text{redactLbl}(\mathcal{G}), j)). \end{aligned} \tag{6}$$

Combining [Equations \(5\)](#) and [\(6\)](#) establishes the invariant via [Equation \(4\)](#).

A.4 Proof of [Theorem 5](#)

Overview. This proof uses the games and adversaries $\mathcal{B}_{\text{INT-PCTXT}}$ and $\mathcal{B}_{\text{CORR-G}}$ in [Fig. 25](#). We establish the following claims.

- $\text{Adv}_{\text{GC}, n, \text{restrict}, \text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$
- $\Pr[\text{bad}_0^{\mathcal{G}_1}] \leq \text{Adv}_{\text{GC}, n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}})$
- $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}]$
- $\Pr[\text{bad}_1^{\mathcal{G}_2}] \leq \text{Adv}_{\text{GC}, n, \text{restrict}, \text{redact}}^{\text{CORR-G}}(\mathcal{B}_{\text{CORR-G}})$
- $\Pr[\mathcal{G}_2] = 0$

Combining the claims gives

$$\begin{aligned}
& \text{Adv}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}}) \\
&= \Pr[\mathcal{G}_0] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\mathcal{G}_1] \\
&\leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\text{bad}_1^{\mathcal{G}_2}] + \Pr[\mathcal{G}_2] \\
&\leq \text{Adv}_{\text{GC},n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}}) \\
&\quad + \text{Adv}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CORR-}\mathcal{G}}(\mathcal{B}_{\text{CORR-}\mathcal{G}}).
\end{aligned}$$

Game $\mathcal{G}_0$. Game $\mathcal{G}_0$ is the CA game augmented with a bookkeeping set S . For each honest send and intended receiver, S stores an entry of the form $(i, j, \text{ctr}_i, m, c)$. This bookkeeping affects neither the oracle responses nor the win flag. Hence $\Pr[\mathcal{G}_0] = \text{Adv}_{\text{GC},n,\text{restrict},\text{redact}}^{\text{CA}}(\mathcal{A}_{\text{CA}})$.

Transition $\mathcal{G}_0 \rightarrow \mathcal{G}_1$. In game $\mathcal{G}_1$, the oracle $\text{RECV}(i, j, c)$ invokes the algorithm GC.Recv . If the algorithm returns $m \neq \perp$, $\mathcal{G}_1$ forms $\text{entry} = (i, j, \text{ctr}, m, c)$. If $\text{entry} \notin S$, the game aborts with output **false**. Games $\mathcal{G}_0$ and $\mathcal{G}_1$ are identical until bad_0 is set. By the Fundamental Lemma, $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$.

To bound $\Pr[\text{bad}_0^{\mathcal{G}_1}]$, $\mathcal{B}_{\text{INT-PCTXT}}$ runs $\mathcal{A}_{\text{CA}}$; see Fig. 25. For each oracle query made by $\mathcal{A}_{\text{CA}}$, $\mathcal{B}_{\text{INT-PCTXT}}$ invokes the matching integrity-game oracle and returns its response unchanged. Until bad_0 becomes true, this reproduces $\mathcal{G}_1$'s oracle interface for $\mathcal{A}_{\text{CA}}$. If $\mathcal{A}_{\text{CA}}$'s RECV query sets bad_0 , $\mathcal{B}_{\text{INT-PCTXT}}$ wins in the INT-PCTXT game because $\text{entry} \notin S$.

Transition $\mathcal{G}_1 \rightarrow \mathcal{G}_2$. Game $\mathcal{G}_2$ omits the assignment $\text{win} \leftarrow \text{true}$ when the receiver's redacted local graph differs from the restriction of the redacted global graph for that receiver. Games $\mathcal{G}_1$ and $\mathcal{G}_2$ are identical until bad_1 is set. By the Fundamental Lemma, $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}]$.

To bound $\Pr[\text{bad}_1^{\mathcal{G}_2}]$, $\mathcal{B}_{\text{CORR-}\mathcal{G}}$ runs $\mathcal{A}_{\text{CA}}$; see Fig. 25. $\mathcal{B}_{\text{CORR-}\mathcal{G}}$ uses the oracles of the CORR- $\mathcal{G}$ game to simulate $\mathcal{G}_2$'s oracle interface for $\mathcal{A}_{\text{CA}}$. Whenever $\mathcal{G}_2$ sets bad_1 , we have $\text{entry} \in S$; otherwise the game would already have aborted. Thus $\mathcal{B}_{\text{CORR-}\mathcal{G}}$ calls the oracle $\text{RECV}(i, j, \text{ctr})$ of the graph-correctness game for the same honest delivery, and that game sets win because the same two resulting graph views differ.

The state and local-graph pairs initially agree because the graph-correctness game supplies its initialized pairs to $\mathcal{B}_{\text{CORR-}\mathcal{G}}$. To answer $\mathcal{A}_{\text{CA}}$'s $\text{SEND}(i, m)$ query, $\mathcal{B}_{\text{CORR-}\mathcal{G}}$'s SIMSEND procedure calls the graph-correctness oracle $\text{SEND}(i, m)$. The SIMSEND procedure stores the returned state and local graph, uses the returned c to record the intended deliveries, and returns $(\mathcal{G}_i^l, c)$ to $\mathcal{A}_{\text{CA}}$, preserving the agreement.

To answer $\mathcal{A}_{\text{CA}}$'s $\text{RECV}(i, j, c)$ query, $\mathcal{B}_{\text{CORR-}\mathcal{G}}$'s SIMRECV procedure runs the deterministic algorithm GC.Recv on $\mathcal{B}_{\text{CORR-}\mathcal{G}}$'s copy of $(st_j, \mathcal{G}_j^l)$. If the algorithm returns $m = \perp$, SIMRECV returns $\perp$ and makes no call to the graph-correctness oracle RECV . State-preserving rejection leaves both pairs unchanged. If $\text{entry} \notin S$, $\mathcal{B}_{\text{CORR-}\mathcal{G}}$ terminates. Otherwise, entry identifies the same ciphertext in the graph-correctness game's table C . The SIMRECV procedure removes entry from S and calls the graph-correctness oracle $\text{RECV}(i, j, \text{ctr})$. That oracle runs the same deterministic algorithm from the same pre-query pair on the same ciphertext, so the updated pairs again agree. The SIMRECV procedure returns $(\mathcal{G}_j^l, m, \text{ctr})$ to $\mathcal{A}_{\text{CA}}$.

Game $\mathcal{G}_2$. Game $\mathcal{G}_2$ never sets win , so $\Pr[\mathcal{G}_2] = 0$.

A.5 Proof of Theorem 6

Overview. This proof uses the games in Fig. 26 and adversaries $\mathcal{B}_{\text{CORR-}m}$ and $\mathcal{B}_{\text{CORR-}\mathcal{G}}$ in Fig. 27. We establish the following claims.

- $\text{Adv}_{\text{GtE},n,\text{restrict},\text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$
- $\Pr[\text{bad}_0^{\mathcal{G}_1}] \leq \text{Adv}_{\text{GC},n}^{\text{CORR-}m}(\mathcal{B}_{\text{CORR-}m})$
- $\Pr[\mathcal{G}_1] \leq \text{Adv}_{\text{GC},n,\text{restrict},\text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{B}_{\text{CORR-}\mathcal{G}})$

Games $\mathcal{G}_0$ – $\mathcal{G}_1$	
$(\text{win}, \text{bad}_0) \leftarrow (\text{false}, \text{false})$	$\text{RECV}(i, j, \text{ctr}) \quad // \quad i, j \in [n], i \neq j, \text{ctr} \in \mathbb{N}$
$(st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l) \leftarrow \$ \text{GC}_G.\text{Init}(n)$	require $C[i, j, \text{ctr}] \neq \perp$
$(st_1^E, \perp), \dots, (st_n^E, \perp) \leftarrow \$ \text{GC}_E.\text{Init}(n)$	$c \leftarrow C[i, j, \text{ctr}]$
$st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$	$(st_j^G, st_j^E) \leftarrow st_j$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$(m, \text{ctr}') \leftarrow (\perp, \perp)$
$\mathcal{G} \leftarrow \text{InitGraph}(n)$	$(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow \text{GC}_E.\text{Recv}(j, st_j^E, \perp, i, c)$
$\mathcal{A}_{\text{CORR-}\mathcal{G}}^{\text{SEND, RECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l)); \text{Return win}$	If $(\text{ctr}_E, \text{payload}) \neq (\text{ctr}, M_E[i, \text{ctr}])$ then
$\text{SEND}(i, m) \quad // \quad i \in [n], m \in \text{GtE.MS}$	$\text{bad}_0 \leftarrow \text{true}$
$\text{ctr}_i \leftarrow \text{ctr}_i + 1$	Return $\perp \quad // \quad \mathcal{G}_{[1, \infty)}$
$(st_i^G, st_i^E) \leftarrow st_i$	If $\text{payload} \neq \perp$ then
$(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \$ \text{GC}_G.\text{Send}(i, st_i^G, \mathcal{G}_i^l, m)$	$(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$
$(st_i^E, \perp, c) \leftarrow \$ \text{GC}_E.\text{Send}(i, st_i^E, \perp, \text{payload})$	If $m \neq \perp$ then
$M_E[i, \text{ctr}_i] \leftarrow \text{payload}$	$st_j \leftarrow (st_j^G, st_j^E); \text{ctr}' \leftarrow \text{ctr}_E$
$st_i \leftarrow (st_i^G, st_i^E)$	If $m = \perp$ then return $\perp$
$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$	$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$
$C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$	If $\text{redactLbl}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redactLbl}(\mathcal{G}), j)$ then
Return $(st_i, \mathcal{G}_i^l, c)$	win $\leftarrow \text{true}$
	$C[i, j, \text{ctr}] \leftarrow \perp; \text{Return } (\mathcal{G}_j^l, m, \text{ctr}')$

Fig. 26: Games $\mathcal{G}_0$ – $\mathcal{G}_1$ for the proof of [Theorem 6](#). The gray-highlighted code is obtained by inlining the bodies of GtE.Init , GtE.Send , and GtE.Recv into the starting game; code added for game transitions is highlighted in green.

Combining the claims gives

$$\begin{aligned}
& \text{Adv}_{\text{GtE}, n, \text{restrict}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{A}_{\text{CORR-}\mathcal{G}}) \\
&= \Pr[\mathcal{G}_0] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\mathcal{G}_1] \\
&\leq \text{Adv}_{\text{GC}_E, n}^{\text{CORR-}m}(\mathcal{B}_{\text{CORR-}m}) \\
&\quad + \text{Adv}_{\text{GC}_G, n, \text{restrict}, \text{redactLbl}}^{\text{CORR-}\mathcal{G}}(\mathcal{B}_{\text{CORR-}\mathcal{G}}).
\end{aligned}$$

Game $\mathcal{G}_0$. Game $\mathcal{G}_0$ is obtained by inlining the algorithms defining GtE in [Fig. 13](#) into the $\text{CORR-}\mathcal{G}$ game for GtE . The table M_E records each payload passed from $\text{GC}_G.\text{Send}$ to $\text{GC}_E.\text{Send}$. The flag bad_0 records the mismatch used for the transition. In $\mathcal{G}_0$, this proof-only bookkeeping affects neither the oracle responses nor the winning condition, establishing the initial-game equality above.

Transition $\mathcal{G}_0 \rightarrow \mathcal{G}_1$. In game $\mathcal{G}_1$, the oracle RECV returns $\perp$ when the algorithm $\text{GC}_E.\text{Recv}$ returns $\text{ctr}_E \neq \text{ctr}$ or $\text{payload} \neq M_E[i, \text{ctr}]$. Games $\mathcal{G}_0$ and $\mathcal{G}_1$ are identical until bad_0 is set, so the Fundamental Lemma establishes the transition bound above. To bound $\Pr[\text{bad}_0^{\mathcal{G}_1}]$, $\mathcal{B}_{\text{CORR-}m}$ runs $\mathcal{A}_{\text{CORR-}\mathcal{G}}$; see [Fig. 27](#). $\mathcal{B}_{\text{CORR-}m}$ receives the initial encryption-layer states from its message-correctness game. It initializes and maintains the graph-layer states and local graphs itself. It therefore has the information needed to reproduce $\mathcal{G}_1$'s oracle interface for $\mathcal{A}_{\text{CORR-}\mathcal{G}}$ until bad_0 is set. On send queries, the oracle SEND returns the updated encryption-layer pair. The oracle RECV does not return the updated encryption-layer state, but the algorithm $\text{GC}_E.\text{Recv}$ is deterministic, so $\mathcal{B}_{\text{CORR-}m}$ can compute that state locally. On receive queries that do not set bad_0 , $\mathcal{B}_{\text{CORR-}m}$ calls the oracle only to keep the message-correctness game's encryption-layer execution synchronized with the simulated execution.

When the algorithm $\text{GC}_E.\text{Recv}$ recovers the counter and payload of the selected honest send, $\mathcal{B}_{\text{CORR-}m}$ calls its oracle RECV only after the algorithm $\text{GC}_G.\text{Recv}$ accepts. The simulated game and the message-correctness game then make the same encryption-layer state update and both record that the selected honest send has been delivered. If the algorithm $\text{GC}_G.\text{Recv}$ rejects, $\mathcal{B}_{\text{CORR-}m}$ discards the encryption-layer update it computed locally, returns $\perp$, and makes no oracle call. If the algorithm $\text{GC}_E.\text{Recv}$ instead returns a counter or payload inconsistent with the selected honest send, bad_0 is set and $\mathcal{B}_{\text{CORR-}m}$ calls its oracle RECV immediately. From the same pre-query state and ciphertext, determinism reproduces the

$\mathcal{B}_{\text{CORR-}m}^{\text{SEND, RECV}}((st_1^E, \perp), \dots, (st_n^E, \perp))$ $(st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l) \leftarrow \text{GC}_G.\text{Init}(n)$ $st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$ $\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$ $\mathcal{A}_{\text{CORR-}G}^{\text{SIMSEND, SIMRECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ Return $\perp$ SIMSEND(i, m) $\text{ctr}_i \leftarrow \text{ctr}_i + 1; (st_i^G, st_i^E) \leftarrow st_i$ $(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \text{GC}_G.\text{Send}(i, st_i^G, \mathcal{G}_i^l, m)$ $(st_i^E, \perp, c) \leftarrow \text{SEND}(i, \text{payload})$ $M_E[i, \text{ctr}_i] \leftarrow \text{payload}; st_i \leftarrow (st_i^G, st_i^E)$ $C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$ Return $(st_i, \mathcal{G}_i^l, c)$ SIMRECV(i, j, ctr) require $C[i, j, \text{ctr}] \neq \perp$ $c \leftarrow C[i, j, \text{ctr}]; (st_j^G, st_j^E) \leftarrow st_j$ $(m, \text{ctr}') \leftarrow (\perp, \perp)$ $(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow \text{GC}_E.\text{Recv}(j, st_j^E, \perp, i, c)$ If $(\text{ctr}_E, \text{payload}) \neq (\text{ctr}, M_E[i, \text{ctr}])$ then RECV (i, j, ctr) ; abort ($\perp$) If $\text{payload} \neq \perp$ then $(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$ If $m \neq \perp$ then RECV (i, j, ctr) $st_j \leftarrow (st_j^G, st_j^E); \text{ctr}' \leftarrow \text{ctr}_E$ If $m = \perp$ then return $\perp$ $C[i, j, \text{ctr}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, m, \text{ctr}')$	$\mathcal{B}_{\text{CORR-}G}^{\text{SEND, RECV}}((st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l))$ $(st_1^E, \perp), \dots, (st_n^E, \perp) \leftarrow \text{GC}_E.\text{Init}(n)$ $st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$ $\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$ $\mathcal{A}_{\text{CORR-}G}^{\text{SIMSEND, SIMRECV}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ Return $\perp$ SIMSEND(i, m) $\text{ctr}_i \leftarrow \text{ctr}_i + 1; (st_i^G, st_i^E) \leftarrow st_i$ $(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \text{SEND}(i, m)$ $(st_i^E, \perp, c) \leftarrow \text{GC}_E.\text{Send}(i, st_i^E, \perp, \text{payload})$ $M_E[i, \text{ctr}_i] \leftarrow \text{payload}; st_i \leftarrow (st_i^G, st_i^E)$ $C[i, j, \text{ctr}_i] \leftarrow c \quad \forall j \in [n] \setminus \{i\}$ Return $(st_i, \mathcal{G}_i^l, c)$ SIMRECV(i, j, ctr) require $C[i, j, \text{ctr}] \neq \perp$ $c \leftarrow C[i, j, \text{ctr}]; (st_j^G, st_j^E) \leftarrow st_j$ $(m, \text{ctr}') \leftarrow (\perp, \perp)$ $(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow \text{GC}_E.\text{Recv}(j, st_j^E, \perp, i, c)$ If $(\text{ctr}_E, \text{payload}) \neq (\text{ctr}, M_E[i, \text{ctr}])$ then return $\perp$ If $\text{payload} \neq \perp$ then $(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$ RECV (i, j, ctr) If $m \neq \perp$ then $st_j \leftarrow (st_j^G, st_j^E); \text{ctr}' \leftarrow \text{ctr}_E$ If $m = \perp$ then return $\perp$ $C[i, j, \text{ctr}] \leftarrow \perp$; Return $(\mathcal{G}_j^l, m, \text{ctr}')$
---	---

Fig. 27: Adversaries $\mathcal{B}_{\text{CORR-}m}$ and $\mathcal{B}_{\text{CORR-}G}$ for the proof of [Theorem 6](#). The adversaries' calls to the oracles of their respective correctness games are highlighted in orange.

inconsistency and sets win. $\mathcal{B}_{\text{CORR-}m}$ then terminates, so no simulation is needed after that query. This establishes the bad-event bound above.

Game $\mathcal{G}_1$. In every receive query that continues in $\mathcal{G}_1$, the algorithm $\text{GC}_E.\text{Recv}$ has recovered the counter and payload associated with the selected honest send. To bound $\Pr[\mathcal{G}_1]$, $\mathcal{B}_{\text{CORR-}G}$ runs $\mathcal{A}_{\text{CORR-}G}$; see [Fig. 27](#). $\mathcal{B}_{\text{CORR-}G}$ receives the graph-layer states and local graphs from its graph-correctness game and initializes the encryption layer itself. It therefore has the information needed to reproduce $\mathcal{G}_1$'s oracle interface. On send queries, the oracle SEND returns the updated graph-layer pair, while $\mathcal{B}_{\text{CORR-}G}$ computes the encryption-layer state itself. The oracle RECV does not return the updated graph-layer state, but the algorithm $\text{GC}_G.\text{Recv}$ is deterministic, so $\mathcal{B}_{\text{CORR-}G}$ can compute that state locally. On continuing receive queries, $\mathcal{B}_{\text{CORR-}G}$ calls the oracle only to keep the graph-layer execution in its graph-correctness game synchronized with the simulated execution.

For a receive query, $\mathcal{B}_{\text{CORR-}G}$ calls its oracle RECV only after the algorithm $\text{GC}_E.\text{Recv}$ recovers the selected counter and payload. Its local evaluation and the oracle's evaluation of the algorithm $\text{GC}_G.\text{Recv}$ then start from the same state and local graph on the same payload. Determinism and state-preserving rejection ensure that the two evaluations either both reject, leaving the receiver's graph-layer state and local graph unchanged, or both accept, make the same update, and record the selected honest send as delivered.

The global graphs in $\mathcal{G}_1$ and the graph-correctness game agree except for labels: $\mathcal{G}_1$ uses (m, c) , whereas the graph-correctness game uses $(m, \text{payload})$. Applying redactLbl therefore gives the same global view in both games, and restrict selects the same target from that view. Every win in $\mathcal{G}_1$ consequently makes $\mathcal{B}_{\text{CORR-}G}$ win in its $\text{CORR-}G$ game, establishing the final claim above.

Games $\mathcal{G}_0$ – $\mathcal{G}_2$	$\text{RECV}(i, j, c) \text{ // } i, j \in [n], i \neq j, c \in \{0, 1\}^*$
$\text{win} \leftarrow \text{false}$	$(st_j^G, st_j^E) \leftarrow st_j$
$(st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l) \leftarrow \text{GC}_G.\text{Init}(n)$	$(m, \text{ctr}) \leftarrow (\perp, \perp)$
$(st_1^E, \perp), \dots, (st_n^E, \perp) \leftarrow \text{GC}_E.\text{Init}(n)$	$(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow \text{GC}_E.\text{Recv}(j, st_j^E, \perp, i, c)$
$st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$	If $\text{payload} \neq \perp$ then
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$e_E \leftarrow (i, j, \text{ctr}_E, \text{payload}, c)$
$(\text{bad}_0, \text{bad}_1, S_E) \leftarrow (\text{false}, \text{false}, \emptyset)$	If $e_E \notin S_E$ then
$S \leftarrow \emptyset; \mathcal{A}_{\text{INT-PCTXT}}^{\text{SEND, RECV}}; \text{Return win}$	$\text{bad}_1 \leftarrow \text{true}$
$\text{SEND}(i, m) \text{ // } i \in [n], m \in \text{GtE.MS}$	$\text{abort}(\text{false}) \text{ // } \mathcal{G}_{[2, \infty)}$
$\text{ctr}_i \leftarrow \text{ctr}_i + 1$	$(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$
$(st_i^G, st_i^E) \leftarrow st_i$	If $\text{bad}_1 = \text{false}$ then
$(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \text{GC}_G.\text{Send}(i, st_i^G, \mathcal{G}_i^l, m)$	If $m \neq M_G[i, \text{ctr}_E]$ then
$M_G[i, \text{ctr}_i] \leftarrow m$	$\text{bad}_0 \leftarrow \text{true}$
$(st_i^E, \perp, c) \leftarrow \text{GC}_E.\text{Send}(i, st_i^E, \perp, \text{payload})$	$\text{abort}(\text{false}) \text{ // } \mathcal{G}_{[1, \infty)}$
$S_E \leftarrow S_E \cup \{(i, j, \text{ctr}_i, \text{payload}, c) \mid j \in [n] \setminus \{i\}\}$	If $m \neq \perp$ then
$st_i \leftarrow (st_i^G, st_i^E)$	$st_j \leftarrow (st_j^G, st_j^E); \text{ctr} \leftarrow \text{ctr}_E$
$S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$	$S_E \leftarrow S_E \setminus \{e_E\}$
Return $(\mathcal{G}_i^l, c)$	If $m = \perp$ then return $\perp$
	$\text{entry} \leftarrow (i, j, \text{ctr}, m, c)$
	If $\text{entry} \notin S$ then $\text{win} \leftarrow \text{true}$ else $S \leftarrow S \setminus \{\text{entry}\}$
	Return $(\mathcal{G}_j^l, m, \text{ctr})$

Fig. 28: Games $\mathcal{G}_0$ – $\mathcal{G}_2$ for the proof of [Theorem 7](#). The gray-highlighted code is obtained by inlining the bodies of GtE.Init , GtE.Send , and GtE.Recv ; code added to form and transition between the proof games is highlighted in **green**.

A.6 Proof of [Theorem 7](#)

Overview. This proof uses the games in [Fig. 28](#) and adversaries $\mathcal{B}_{\text{CORR-}m}$ and $\mathcal{B}_{\text{INT-PCTXT}}$ in [Fig. 29](#). We establish the following claims.

- $\text{Adv}_{\text{GtE}, n}^{\text{INT-PCTXT}}(\mathcal{A}_{\text{INT-PCTXT}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$
- $\Pr[\text{bad}_0^{\mathcal{G}_1}] \leq \text{Adv}_{\text{GC}_G, n}^{\text{CORR-}m}(\mathcal{B}_{\text{CORR-}m})$
- $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}]$
- $\Pr[\text{bad}_1^{\mathcal{G}_2}] \leq \text{Adv}_{\text{GC}_E, n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}})$
- $\Pr[\mathcal{G}_2] = 0$

Combining the claims gives

$$\begin{aligned}
& \text{Adv}_{\text{GtE}, n}^{\text{INT-PCTXT}}(\mathcal{A}_{\text{INT-PCTXT}}) \\
&= \Pr[\mathcal{G}_0] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\mathcal{G}_1] \\
&\leq \text{Adv}_{\text{GC}_G, n}^{\text{CORR-}m}(\mathcal{B}_{\text{CORR-}m}) + \Pr[\text{bad}_1^{\mathcal{G}_2}] + \Pr[\mathcal{G}_2] \\
&\leq \text{Adv}_{\text{GC}_G, n}^{\text{CORR-}m}(\mathcal{B}_{\text{CORR-}m}) + \text{Adv}_{\text{GC}_E, n}^{\text{INT-PCTXT}}(\mathcal{B}_{\text{INT-PCTXT}}).
\end{aligned}$$

Game $\mathcal{G}_0$. Game $\mathcal{G}_0$ is obtained by inlining the algorithms defining GtE in [Fig. 13](#) into the INT-PCTXT game for GtE . The inherited set S records the outer GtE tuples tested by that game's winning condition. The proof-added set S_E records the corresponding encryption-layer tuples. The table M_G records the messages supplied to the algorithm $\text{GC}_G.\text{Send}$. The persistent flag bad_1 records whether an encryption-layer tuple absent from S_E has occurred. Before this flag is set, bad_0 records a graph-layer message mismatch. These flags and the added bookkeeping affect neither the oracle responses nor win , so $\mathcal{G}_0$ is equivalent to the INT-PCTXT game for GtE . This establishes the first claim. The reductions below omit the outer set S because it affects neither the oracle responses nor the two bad events.

$\mathcal{B}_{\text{CORR-}m}^{\text{SEND, RECV}}((st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l))$ $(st_1^E, \perp), \dots, (st_n^E, \perp) \leftarrow \text{GC}_E.\text{Init}(n)$ $st_1 \leftarrow (st_1^G, st_1^E), \dots, st_n \leftarrow (st_n^G, st_n^E)$ $\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$ $S_E \leftarrow \emptyset; \mathcal{A}_{\text{INT-PCTXT}}^{\text{SIMSEND, SIMRECV}}; \text{Return } \perp$ $\text{SIMSEND}(i, m)$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1; (st_i^G, st_i^E) \leftarrow st_i$ $(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \text{SEND}(i, m)$ $M_G[i, \text{ctr}_i] \leftarrow m$ $(st_i^E, \perp, c) \leftarrow \text{GC}_E.\text{Send}(i, st_i^E, \perp, \text{payload})$ $S_E \leftarrow S_E \cup \{(i, j, \text{ctr}_i, \text{payload}, c) \mid j \in [n] \setminus \{i\}\}$ $st_i \leftarrow (st_i^G, st_i^E); \text{Return } (\mathcal{G}_i^l, c)$ $\text{SIMRECV}(i, j, c)$ $(st_j^G, st_j^E) \leftarrow st_j; (m, \text{ctr}) \leftarrow (\perp, \perp)$ $(st_j^E, \perp, \text{payload}, \text{ctr}_E) \leftarrow \text{GC}_E.\text{Recv}(j, st_j^E, \perp, i, c)$ If $\text{payload} \neq \perp$ then $e_E \leftarrow (i, j, \text{ctr}_E, \text{payload}, c)$ If $e_E \notin S_E$ then abort ($\perp$) $(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$ $\text{RECV}(i, j, \text{ctr}_E)$ If $m \neq M_G[i, \text{ctr}_E]$ then abort ($\perp$) If $m \neq \perp$ then $st_j \leftarrow (st_j^G, st_j^E); \text{ctr} \leftarrow \text{ctr}_E$ $S_E \leftarrow S_E \setminus \{e_E\}$ If $m = \perp$ then return $\perp$ Return $(\mathcal{G}_j^l, m, \text{ctr})$	$\mathcal{B}_{\text{INT-PCTXT}}^{\text{SEND, RECV}}$ $(st_1^G, \mathcal{G}_1^l), \dots, (st_n^G, \mathcal{G}_n^l) \leftarrow \text{GC}_G.\text{Init}(n)$ $\text{ctr}_i \leftarrow 0 \quad \forall i \in [n]$ $S_E \leftarrow \emptyset; \mathcal{A}_{\text{INT-PCTXT}}^{\text{SIMSEND, SIMRECV}}; \text{Return } \perp$ $\text{SIMSEND}(i, m)$ $\text{ctr}_i \leftarrow \text{ctr}_i + 1$ $(st_i^G, \mathcal{G}_i^l, \text{payload}) \leftarrow \text{GC}_G.\text{Send}(i, st_i^G, \mathcal{G}_i^l, m)$ $M_G[i, \text{ctr}_i] \leftarrow m$ $(\perp, c) \leftarrow \text{SEND}(i, \text{payload})$ $S_E \leftarrow S_E \cup \{(i, j, \text{ctr}_i, \text{payload}, c) \mid j \in [n] \setminus \{i\}\}$ Return $(\mathcal{G}_i^l, c)$ $\text{SIMRECV}(i, j, c)$ $(m, \text{ctr}) \leftarrow (\perp, \perp)$ $z \leftarrow \text{RECV}(i, j, c)$ If $z \neq \perp$ then $(\perp, \text{payload}, \text{ctr}_E) \leftarrow z$ $e_E \leftarrow (i, j, \text{ctr}_E, \text{payload}, c)$ If $e_E \notin S_E$ then abort ($\perp$) $(st_j^G, \mathcal{G}_j^l, m, \text{ctr}_G) \leftarrow \text{GC}_G.\text{Recv}(j, st_j^G, \mathcal{G}_j^l, i, \text{payload})$ If $m \neq M_G[i, \text{ctr}_E]$ then abort ($\perp$) If $m \neq \perp$ then $\text{ctr} \leftarrow \text{ctr}_E; S_E \leftarrow S_E \setminus \{e_E\}$ If $m = \perp$ then return $\perp$ Return $(\mathcal{G}_j^l, m, \text{ctr})$
--	--

Fig. 29: Adversaries $\mathcal{B}_{\text{CORR-}m}$ and $\mathcal{B}_{\text{INT-PCTXT}}$ for the proof of Theorem 7. The adversaries' calls to the oracles of their respective security games are highlighted in orange.

Transition $\mathcal{G}_0 \rightarrow \mathcal{G}_1$. Game $\mathcal{G}_1$ aborts with output false when, before bad_1 has been set, the algorithm $\text{GC}_G.\text{Recv}$ returns $m \neq M_G[i, \text{ctr}_E]$, including $m = \perp$. This condition sets bad_0 . Games $\mathcal{G}_0$ and $\mathcal{G}_1$ are identical until bad_0 is set, so the Fundamental Lemma gives

$$\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}].$$

To bound $\Pr[\text{bad}_0^{\mathcal{G}_1}]$, $\mathcal{B}_{\text{CORR-}m}$ runs $\mathcal{A}_{\text{INT-PCTXT}}$; see Fig. 29. $\mathcal{B}_{\text{CORR-}m}$ uses the oracles of the message-correctness game for GC_G to supply the graph-layer behavior and maintains the encryption layer locally. It returns the same responses as $\mathcal{G}_1$ for every query completed before either bad_0 or bad_1 is set. On send queries, the oracle SEND returns the updated graph-layer state and local graph. On receive queries, determinism lets $\mathcal{B}_{\text{CORR-}m}$ evaluate the algorithm $\text{GC}_G.\text{Recv}$ locally before invoking its oracle RECV on the corresponding honest send.

If $e_E = (i, j, \text{ctr}_E, \text{payload}, c)$ is absent from S_E , $\mathcal{G}_1$ sets bad_1 . The persistent guard then prevents $\mathcal{G}_1$ from setting bad_0 on this or any later query, so $\mathcal{B}_{\text{CORR-}m}$ may terminate. Otherwise, e_E identifies the honest graph-layer send indexed by ctr_E . The locally evaluated algorithm and the oracle RECV therefore process the same payload from the same pre-query state. A return value different from $M_G[i, \text{ctr}_E]$, including $\perp$, makes $\mathcal{B}_{\text{CORR-}m}$ win in its message-correctness game under the condition that sets bad_0 in $\mathcal{G}_1$. On every continuing query, both executions make the same graph-layer update. This establishes the bad_0 bound above. The message-correctness game also sets its win flag if $\text{ctr}_G \neq \text{ctr}_E$; this additional way to win only loosens the bound.

Transition $\mathcal{G}_1 \rightarrow \mathcal{G}_2$. Game $\mathcal{G}_2$ additionally aborts with output false when $e_E \notin S_E$, which is the condition that sets bad_1 . Games $\mathcal{G}_1$ and $\mathcal{G}_2$ are identical until bad_1 is set, so the Fundamental Lemma gives

$$\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}].$$

To bound $\Pr[\text{bad}_1^{\mathcal{G}_2}]$, $\mathcal{B}_{\text{INT-PCTXT}}$ runs $\mathcal{A}_{\text{INT-PCTXT}}$; see Fig. 29. $\mathcal{B}_{\text{INT-PCTXT}}$ uses the oracles of the INT-PCTXT game for GC_E to supply the encryption-layer behavior and maintains the graph layer locally. After every simulated oracle call that returns to $\mathcal{A}_{\text{INT-PCTXT}}$, $\mathcal{B}_{\text{INT-PCTXT}}$'s local set S_E equals the set S in its encryption-layer INT-PCTXT game. Consequently, $\mathcal{B}_{\text{INT-PCTXT}}$ gives $\mathcal{A}_{\text{INT-PCTXT}}$ the same oracle response whenever $\mathcal{G}_2$ does not abort on the query. On send queries, $\mathcal{B}_{\text{INT-PCTXT}}$ evaluates the graph layer, passes the resulting payload to its oracle SEND , and records the corresponding entry in S_E .

On a receive query, $\mathcal{B}_{\text{INT-PCTXT}}$ calls its encryption-layer oracle RECV , which supplies the payload and counter. If the oracle rejects, the simulated query also rejects. If the resulting e_E is absent from S_E , the same oracle call makes $\mathcal{B}_{\text{INT-PCTXT}}$ win under the condition that sets bad_1 in $\mathcal{G}_2$, and $\mathcal{B}_{\text{INT-PCTXT}}$ terminates. Otherwise, $\mathcal{B}_{\text{INT-PCTXT}}$ evaluates the algorithm $\text{GC}_G.\text{Recv}$. If the returned message differs from $M_G[i, \text{ctr}_E]$, $\mathcal{G}_2$ aborts with output `false` and $\mathcal{B}_{\text{INT-PCTXT}}$ terminates. After every continuing receive query, $\mathcal{B}_{\text{INT-PCTXT}}$ updates its graph-layer pair and S_E and returns the same response as $\mathcal{G}_2$. This establishes the bad_1 bound above.

Game $\mathcal{G}_2$. Every execution that reaches the inherited test entry $\notin S$ has triggered neither abort in $\mathcal{G}_2$. Thus e_E identifies an honest encryption-layer send, and the returned message equals the message recorded in M_G for the corresponding graph-layer send. The sets S and S_E are populated together, and every earlier non-aborting accepted receive removes the corresponding entries from both sets. Hence, when neither abort occurs, entry $= (i, j, \text{ctr}, m, c)$ belongs to S . The inherited assignment $\text{win} \leftarrow \text{true}$ is therefore unreachable in $\mathcal{G}_2$, establishing the final claim above.

B Checkpoint-aided proofs (Proof of Theorem 8)

Overview. This proof uses the games in Fig. 30 and adversaries $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ and $\mathcal{B}_{\text{CORR-G-chkpt}}$. Their pseudocode appears in Figs. 31 and 32. We establish the following claims.

- $\text{Adv}_{\text{GC}_{\text{CP}}, n, \text{restrict}, \text{redact}}^{\text{CA-chkpt}}(\mathcal{A}_{\text{CA-chkpt}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$
- $\Pr[\text{bad}_0^{\mathcal{G}_1}] \leq \text{Adv}_{\text{GC}_{\text{CP}}, n}^{\text{INT-PCTXT-chkpt}}(\mathcal{B}_{\text{INT-PCTXT-chkpt}})$
- $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}]$
- $\Pr[\text{bad}_1^{\mathcal{G}_2}] \leq \text{Adv}_{\text{GC}_{\text{CP}}, n, \text{restrict}, \text{redact}}^{\text{CORR-G-chkpt}}(\mathcal{B}_{\text{CORR-G-chkpt}})$
- $\Pr[\mathcal{G}_2] = 0$.

Proof sketch. Game $\mathcal{G}_0$ records honest sends and epoch state without changing the causal-awareness game. Game $\mathcal{G}_1$ aborts whenever bad_0 is set. This happens if the algorithm invoked by RECV or APPLYCHKPT accepts without a matching honest SEND or CREATECHKPT call, or if $\text{GC}_{\text{CP}}.\text{Recv}$ reports an unopened epoch. Adversary $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ answers each query using the matching checkpoint-integrity oracle, so every bad_0 event makes $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ win in that game. Thus the probability of bad_0 is bounded by its INT-PCTXT-chkpt advantage. Game $\mathcal{G}_2$ suppresses wins caused when the receiver's redacted local graph differs from the receiver's required view of the redacted global graph, and rejects reused transcript hashes. Both changes occur only after bad_1 is set. If $\mathcal{G}_1$ does not abort, every accepting Recv or ApplyChkPt evaluation matches an unused entry created by an honest SEND or CREATECHKPT call. Adversary $\mathcal{B}_{\text{CORR-G-chkpt}}$ evaluates the corresponding deterministic algorithm locally and invokes the corresponding graph-correctness oracle. This matching, together with state preservation on rejection, keeps the receiver state, local graph, and epoch bookkeeping synchronized with the graph-correctness game until bad_1 is set. Thus every bad_1 event makes $\mathcal{B}_{\text{CORR-G-chkpt}}$ win. The Fundamental Lemma bounds both hops, and $\mathcal{G}_2$ has no remaining win condition.

Game $\mathcal{G}_0$. Game $\mathcal{G}_0$ is the CA-chkpt game augmented with the code highlighted in green. The set S stores the entries for honest message and checkpoint sends, while openEpochs_j and usedEpochs_j track receiver epochs. The added code records these values and sets bad_0 and bad_1 when the corresponding events

Games $\mathcal{G}_0$–$\mathcal{G}_2$ (win, bad₀, bad₁) $\leftarrow$ (false, false, false) $\mathcal{G} \leftarrow \text{InitGraph}(n)$ ($\text{ctr}_j^{\text{in-epoch}}, \text{th}_j$) $\leftarrow$ (0, ϵ) $\forall j \in [n]$ (openEpochs_j, usedEpochs_j) $\leftarrow$ ({ϵ}, {ϵ}) $\forall j \in [n]$ $S \leftarrow \emptyset$; ($st_1, \mathcal{G}_1^l$), ..., ($st_n, \mathcal{G}_n^l$) $\leftarrow$ $\text{GC}_{\text{CP}}.\text{Init}(n)$ $\mathcal{A}_{\text{CA-chkpt}}^{\text{SEND, RECV, CREATECHKPT, APPLYCHKPT}}$; Return win SEND($i, m$) // $i \in [n], m \in \text{GC}_{\text{CP}}.\text{MS}$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ ($st_i, \mathcal{G}_i^l, c$) $\leftarrow$ $\text{GC}_{\text{CP}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{MSG}, m, c))$ msgInfo $\leftarrow$ (MSG, $\text{th}_i, \text{ctr}_i^{\text{in-epoch}}, m$) $S \leftarrow S \cup \{(i, j, \text{msgInfo}, c) \mid j \in [n] \setminus \{i\}\}$ Return ($\mathcal{G}_i^l, c$) RECV(i, j, c) // $i, j \in [n], i \neq j, c \in \{0, 1\}^*$ ($st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}$) $\leftarrow$ $\text{GC}_{\text{CP}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ If $\text{th} \notin \text{openEpochs}_j$ then bad₀ $\leftarrow$ true // $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ wins. abort(false) // $\mathcal{G}_{[1, \infty]}$ msgInfo $\leftarrow$ (MSG, $\text{th}, \text{ctr}_j^{\text{in-epoch}}, m$) entry $\leftarrow$ ($i, j, \text{msgInfo}, c$) If entry $\notin S$ then bad₀ $\leftarrow$ true // $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ wins. abort(false) // $\mathcal{G}_{[1, \infty]}$ Else $S \leftarrow S \cup \{\text{entry}\}$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}, \text{ctr}_j^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then bad₁ $\leftarrow$ true // $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ wins. win $\leftarrow$ true // $\mathcal{G}_{[0, 2]}$ Return ($\mathcal{G}_j^l, m, \text{th}, \text{ctr}_j^{\text{in-epoch}}$)	CREATECHKPT(i) // $i \in [n]$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ ($st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}$) $\leftarrow$ $\text{GC}_{\text{CP}}.\text{CreateChkPt}(i, st_i, \mathcal{G}_i^l)$ $\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, (\text{th}_i, \text{ctr}_i^{\text{in-epoch}}), (\text{CHKPT}, \text{th}^{\text{new}}, c))$ chkptInfo $\leftarrow$ (CHKPT, $\text{th}_i, \text{ctr}_i^{\text{in-epoch}}, \text{th}^{\text{new}}$) $S \leftarrow S \cup \{(i, j, \text{chkptInfo}, c) \mid j \in [n]\}$ Return ($\mathcal{G}_i^l, c, \text{th}^{\text{new}}$) APPLYCHKPT($i, j, c$) // $i, j \in [n], c \in \{0, 1\}^*$ ($st_j, \mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}$) $\leftarrow$ $\text{GC}_{\text{CP}}.\text{ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{th}^{\text{new}} = \perp$ then return $\perp$ chkptInfo $\leftarrow$ (CHKPT, $\text{th}_j, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}$) entry $\leftarrow$ ($i, j, \text{chkptInfo}, c$) If entry $\notin S$ then bad₀ $\leftarrow$ true // $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ wins. abort(false) // $\mathcal{G}_{[1, \infty]}$ Else $S \leftarrow S \cup \{\text{entry}\}$ If $\text{th}^{\text{new}} \in \text{usedEpochs}_j$ then bad₁ $\leftarrow$ true // $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ wins. Return $\perp$ // $\mathcal{G}_{[2, \infty]}$ $\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, (\text{th}_j, \text{ctr}_j^{\text{in-epoch}}))$ If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then bad₁ $\leftarrow$ true // $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ wins. win $\leftarrow$ true // $\mathcal{G}_{[0, 2]}$ $\text{th}_j \leftarrow \text{th}^{\text{new}}$ openEpochs_j $\leftarrow$ openEpochs_j $\cup$ {th^{new}} usedEpochs_j $\leftarrow$ usedEpochs_j $\cup$ {th^{new}} $\text{ctr}_j^{\text{in-epoch}} \leftarrow 0$ Return ($\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}}$)
--	---

Fig. 30: Games $\mathcal{G}_0$ – $\mathcal{G}_2$ for the proof of Theorem 8. Code added for game transitions is highlighted in green.

occur. The added code changes neither an oracle output nor the winning condition of $\mathcal{G}_0$, establishing the initial-game equality above.

Transition $\mathcal{G}_0 \rightarrow \mathcal{G}_1$. Game $\mathcal{G}_1$ aborts with output false under two added conditions. The oracle $\text{RECV}(i, j, c)$ invokes the algorithm $\text{GC}_{\text{CP}}.\text{Recv}$. If the algorithm returns $m \neq \perp$ and either $\text{th} \notin \text{openEpochs}_j$ or $\text{entry} \notin S$, the game aborts. The oracle $\text{APPLYCHKPT}(i, j, c)$ invokes the algorithm $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$. If the algorithm returns $\text{th}^{\text{new}} \neq \perp$ and $\text{entry} \notin S$, the game aborts. Games $\mathcal{G}_0$ and $\mathcal{G}_1$ are identical until bad_0 is set, so the Fundamental Lemma establishes the transition bound above. For the associated bad-event bound, $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ runs $\mathcal{A}_{\text{CA-chkpt}}$ and invokes the matching checkpoint-integrity oracle for each query. Until bad_0 is set, this reproduces $\mathcal{G}_1$'s oracle interface for $\mathcal{A}_{\text{CA-chkpt}}$. Whenever a query sets bad_0 in $\mathcal{G}_1$, the corresponding checkpoint-integrity call lets $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ win in the INT-PCTXT-chkpt game.

Transition $\mathcal{G}_1 \rightarrow \mathcal{G}_2$. Game $\mathcal{G}_2$ suppresses wins caused when the receiver's redacted local graph differs from the receiver's required view of the redacted global graph. It also makes APPLYCHKPT return $\perp$ when an accepted checkpoint application produces a previously used transcript hash. In both cases, bad_1 is set before the games diverge, so the Fundamental Lemma establishes the fourth claim. For the fifth claim, once bad_0 is excluded, each accepted receive-side query identifies an entry created by a matching honest Send or CreateChkPt call. Adversary $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ then invokes the corresponding graph-correctness

Adversary $\mathcal{B}_{\text{INT-PCTXT-chkpt}}^{\text{SEND, RECV, CREATECHKPT, APPLYCHKPT}}$ $\mathcal{A}_{\text{CA-chkpt}}^{\text{SIMSEND, SIMRECV, SIMCREATECHKPT, SIMAPPLYCHKPT}}$ Return $\perp$	
$\text{SIMSEND}(i, m)$ Return $\text{SEND}(i, m)$	$\text{SIMCREATECHKPT}(i)$ Return $\text{CREATECHKPT}(i)$
$\text{SIMRECV}(i, j, c)$ Return $\text{RECV}(i, j, c)$	$\text{SIMAPPLYCHKPT}(i, j, c)$ Return $\text{APPLYCHKPT}(i, j, c)$

Fig. 31: Adversary $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ for the proof of [Theorem 8](#). The adversary's calls to the oracles of the checkpoint-integrity game are highlighted in **orange**.

Adversary $\mathcal{B}_{\text{CORR-G-chkpt}}^{\text{O}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$ $S \leftarrow \emptyset$; $\text{eventCtrOf} \leftarrow \emptyset$ $(\text{ctr}_j^{\text{in-epoch}}, \text{th}_j) \leftarrow (0, \epsilon) \quad \forall j \in [n]$ $(\text{openEpochs}_j, \text{usedEpochs}_j) \leftarrow (\{\epsilon\}, \{\epsilon\}) \quad \forall j \in [n]$ $\mathcal{A}_{\text{CA-chkpt}}^{\text{SIMSEND, SIMRECV, SIMCREATECHKPT, SIMAPPLYCHKPT}}$; Return $\perp$ $\text{SIMSEND}(i, m)$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{ctr}_i^{\text{game}}) \leftarrow \text{SEND}(i, m)$ $\text{msgInfo} \leftarrow (\text{MSG}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, m)$ $S \leftarrow S \cup \{(i, j, \text{msgInfo}, c) \mid j \in [n] \setminus \{i\}\}$ $\text{eventCtrOf}[i, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}] \leftarrow \text{ctr}_i^{\text{game}}$; Return $(\mathcal{G}_i^l, c)$ $\text{SIMRECV}(i, j, c)$ $(st_j, \mathcal{G}_j^l, m, \text{th}, \text{ctr}^{\text{in-epoch}}) \leftarrow \text{GC}_{\text{CP}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$ If $m = \perp$ then return $\perp$ If $\text{th} \notin \text{openEpochs}_j$ then abort ($\perp$) $\text{msgInfo} \leftarrow (\text{MSG}, \text{th}, \text{ctr}^{\text{in-epoch}}, m)$ $\text{entry} \leftarrow (i, j, \text{msgInfo}, c)$ If $\text{entry} \notin S$ then abort ($\perp$) $S \leftarrow S \setminus \{\text{entry}\}$; $\text{ctr}^{\text{game}} \leftarrow \text{eventCtrOf}[i, \text{th}, \text{ctr}^{\text{in-epoch}}]$ $\text{RECV}(i, j, \text{ctr}^{\text{game}})$ Return $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}^{\text{in-epoch}})$	$\text{SIMCREATECHKPT}(i)$ $\text{ctr}_i^{\text{in-epoch}} \leftarrow \text{ctr}_i^{\text{in-epoch}} + 1$ $(st_i, \mathcal{G}_i^l, c, \text{th}^{\text{new}}, \text{ctr}_i^{\text{game}}) \leftarrow \text{CREATECHKPT}(i)$ $\text{chkptInfo} \leftarrow (\text{CHKPT}, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}, \text{th}^{\text{new}})$ $S \leftarrow S \cup \{(i, j, \text{chkptInfo}, c) \mid j \in [n]\}$ $\text{eventCtrOf}[i, \text{th}_i, \text{ctr}_i^{\text{in-epoch}}] \leftarrow \text{ctr}_i^{\text{game}}$; Return $(\mathcal{G}_i^l, c, \text{th}^{\text{new}})$ $\text{SIMAPPLYCHKPT}(i, j, c)$ $(st_j, \mathcal{G}_j^l, \text{ctr}^{\text{in-epoch}}, \text{th}^{\text{new}}) \leftarrow \text{GC}_{\text{CP}}.\text{ApplyChkPt}(j, st_j, \mathcal{G}_j^l, i, c)$ If $\text{th}^{\text{new}} = \perp$ then return $\perp$ $\text{chkptInfo} \leftarrow (\text{CHKPT}, \text{th}_j, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$ $\text{entry} \leftarrow (i, j, \text{chkptInfo}, c)$ If $\text{entry} \notin S$ then abort ($\perp$) $S \leftarrow S \setminus \{\text{entry}\}$; $\text{ctr}^{\text{game}} \leftarrow \text{eventCtrOf}[i, \text{th}_j, \text{ctr}_j^{\text{in-epoch}}]$ $\text{APPLYCHKPT}(i, j, \text{ctr}^{\text{game}})$ If $\text{th}^{\text{new}} \in \text{usedEpochs}_j$ then abort ($\perp$) $\text{th}_j \leftarrow \text{th}^{\text{new}}$; $\text{ctr}_j^{\text{in-epoch}} \leftarrow 0$ $\text{openEpochs}_j \leftarrow \text{openEpochs}_j \cup \{\text{th}^{\text{new}}\}$ $\text{usedEpochs}_j \leftarrow \text{usedEpochs}_j \cup \{\text{th}^{\text{new}}\}$ Return $(\mathcal{G}_j^l, \text{ctr}_j^{\text{in-epoch}}, \text{th}^{\text{new}})$
--	---

Fig. 32: Adversary $\mathcal{B}_{\text{CORR-G-chkpt}}$ for the proof of [Theorem 8](#). Here $\text{O} = \{\text{SEND, RECV, CREATECHKPT, APPLYCHKPT}\}$. The adversary's calls to the oracles of the graph-correctness game are highlighted in **orange**.

oracle. Until bad_1 , determinism and state preservation on algorithm rejection keep the receiver state, local graph, and epoch bookkeeping synchronized, so every event setting bad_1 also wins the graph-correctness game.

Game $\mathcal{G}_2$. Game $\mathcal{G}_2$ contains no reachable win condition, establishing the final claim above. **Integrity adversary.** Adversary $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ runs $\mathcal{A}_{\text{CA-chkpt}}$; see [Fig. 31](#). For each oracle query made by $\mathcal{A}_{\text{CA-chkpt}}$, $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ invokes the matching checkpoint-integrity oracle and returns its response unchanged. Until bad_0 is set, this reproduces $\mathcal{G}_1$'s oracle interface for $\mathcal{A}_{\text{CA-chkpt}}$. This direct simulation gives the checkpoint-integrity bound stated above. Whenever a query made by $\mathcal{A}_{\text{CA-chkpt}}$ sets bad_0 in $\mathcal{G}_1$, the identically named checkpoint-integrity call lets $\mathcal{B}_{\text{INT-PCTXT-chkpt}}$ win in the INT-PCTXT-chkpt game.

Graph-correctness adversary. Adversary $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ runs $\mathcal{A}_{\text{CA-chkpt}}$; see Fig. 32. $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ uses the graph-correctness game’s oracles to simulate $\mathcal{G}_2$ ’s oracle interface for $\mathcal{A}_{\text{CA-chkpt}}$ until bad_1 is set. This simulation gives the graph-correctness bound stated above.

$\text{eventCtrOf}[i, \tau, a]$ maps sender i ’s send-event identifier (τ, a) to the game-only event counter returned by the corresponding oracle of the graph-correctness game, SEND or CREATECHKPT .

To answer $\mathcal{A}_{\text{CA-chkpt}}$ ’s RECV queries, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s SIMRECV procedure first runs the algorithm $\text{GC}_{\text{CP}}.\text{Recv}$ on $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s stored receiver state and local graph. To answer $\mathcal{A}_{\text{CA-chkpt}}$ ’s APPLYCHKPT queries, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ uses SIMAPPLYCHKPT . The SIMAPPLYCHKPT procedure first runs the algorithm $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$ on the same stored values. Both algorithms are deterministic, and rejection preserves these inputs.

To answer $\mathcal{A}_{\text{CA-chkpt}}$ ’s SEND query, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s SIMSEND procedure calls the graph-correctness oracle SEND . The SIMSEND procedure stores the returned state and local graph, records the honest-send data, stores the event counter in eventCtrOf , and returns $(\mathcal{G}_i^l, c)$ to $\mathcal{A}_{\text{CA-chkpt}}$. To answer $\mathcal{A}_{\text{CA-chkpt}}$ ’s CREATECHKPT query, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s SIMCREATECHKPT procedure calls the graph-correctness oracle CREATECHKPT . The SIMCREATECHKPT procedure stores the returned state and local graph, records the checkpoint data, stores the event counter in eventCtrOf , and returns $(\mathcal{G}_i^l, c, \text{th}^{\text{new}})$ to $\mathcal{A}_{\text{CA-chkpt}}$. The SIMRECV and SIMAPPLYCHKPT procedures return $\perp$ if the corresponding algorithm, $\text{GC}_{\text{CP}}.\text{Recv}$ or $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$, rejects. If the algorithm $\text{GC}_{\text{CP}}.\text{Recv}$ accepts, SIMRECV aborts unless $\text{th} \in \text{openEpochs}_j$ and $\text{entry} \in S$. If the algorithm $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$ accepts, SIMAPPLYCHKPT aborts unless $\text{entry} \in S$. These are the forwarding conditions. When they hold, the matching entry in S identifies the corresponding honest graph send vertex. The eventCtrOf lookup then returns that vertex’s graph-correctness event counter for the ensuing call to the graph-correctness oracle RECV or APPLYCHKPT . After the graph-correctness oracle RECV returns, SIMRECV returns $(\mathcal{G}_j^l, m, \text{th}, \text{ctr}^{\text{in-epoch}})$ to $\mathcal{A}_{\text{CA-chkpt}}$. After the graph-correctness oracle APPLYCHKPT returns with a fresh transcript hash, SIMAPPLYCHKPT updates the epoch bookkeeping and returns $(\mathcal{G}_j^l, \text{ctr}^{\text{in-epoch}}, \text{th}^{\text{new}})$ to $\mathcal{A}_{\text{CA-chkpt}}$.

Because the forwarding conditions hold, the event metadata returned by the corresponding receive-side algorithm equals the honest metadata stored for the selected send vertex. Consequently, whenever $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ calls the graph-correctness oracle RECV or APPLYCHKPT , the graph-correctness game compares the receiver’s redacted local graph with the restriction of the redacted global graph. On an APPLYCHKPT call, the game also checks whether a transcript hash is reused. These are also the conditions that set bad_1 in $\mathcal{G}_2$.

More precisely, when SIMRECV begins, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s stored $(st_j, \mathcal{G}_j^l)$ equals the corresponding pair maintained by the graph-correctness game for every j . The same invariant holds when SIMAPPLYCHKPT begins. The transcript-hash values, action counters, and epoch sets stored by $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ also equal the corresponding values maintained by the graph-correctness game. The initial values agree because the graph-correctness game supplies the state and graph pairs to $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$, which initializes the same counter and epoch values. When $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ answers $\mathcal{A}_{\text{CA-chkpt}}$ ’s SEND or CREATECHKPT query, the send-side steps described above preserve this invariant. If $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ ’s evaluation of either algorithm, $\text{GC}_{\text{CP}}.\text{Recv}$ or $\text{GC}_{\text{CP}}.\text{ApplyChkPt}$, rejects, the state and local graph remain unchanged. In that case, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ makes no corresponding call to the oracle RECV or APPLYCHKPT of the graph-correctness game. If a forwarding condition fails, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ terminates. After either algorithm accepts and the forwarding conditions hold, the corresponding graph-correctness oracle evaluates the same deterministic algorithm from the matching pre-query pair on the same ciphertext. After the graph-correctness oracle RECV returns, the two state pairs agree again. They also agree after the graph-correctness oracle APPLYCHKPT returns with a fresh transcript hash. $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ has removed the corresponding entry from S , while the graph-correctness game clears the corresponding entry of the ciphertext table C . On a checkpoint application with a fresh transcript hash, $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ and the graph-correctness game also update the same transcript-hash and action-counter values and the same epoch sets. If a call to the graph-correctness oracle APPLYCHKPT reuses a transcript hash, the graph-correctness game sets win. $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ then aborts, so the invariant need not be re-established after that event.

Thus $\mathcal{B}_{\text{CORR-}\mathcal{G}\text{-chkpt}}$ makes every graph-correctness RECV or APPLYCHKPT call from matching pre-query receiver states and local graphs. Each call uses the graph-correctness event counter returned for the matching honest send event. The graph-correctness game therefore sets win under the same two

Games $\mathcal{G}_0$ – $\mathcal{G}_2$	
$\text{win} \leftarrow \text{false}; \mathcal{G} \leftarrow \text{InitGraph}(n)$	$\text{SEND}(i, m) \quad // \quad i \in [n], m \in \text{GC}_{\text{SA}}.\text{MS}$
$(\text{bad}_0, \text{bad}_1, S) \leftarrow (\text{false}, \text{false}, \emptyset)$	$\text{ctr}_i \leftarrow \text{ctr}_i + 1$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{GC}_{\text{SA}}.\text{Send}(i, st_i, \mathcal{G}_i^l, m)$
$(st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l) \leftarrow \text{GC}_{\text{SA}}.\text{Init}(n)$	$\text{sendEvidence}[i, \text{ctr}_i] \leftarrow \text{evidence}$
$\mathcal{A}_{\text{CA-SA}}^{\text{SEND, RECV, VERIFY}}$	$S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$
Return win	$\mathcal{G} \leftarrow \text{AddSendVrtx}(\mathcal{G}, i, \text{ctr}_i, (m, c))$
$\text{RECV}(i, j, c) \quad // \quad i, j \in [n], i \neq j, c \in \{0, 1\}^*$	Return $(\mathcal{G}_i^l, c, \text{evidence})$
$(st_j, \mathcal{G}_j^l, m, \text{ctr}, \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$	$\text{VERIFY}(i, j, \text{ctr}, \text{evidence}')$
If $m = \perp$ then return $\perp$	$// \quad i, j \in [n], \text{ctr} \in \mathbb{N}, \text{evidence}' \in \{0, 1\}^*$
$\text{entry} \leftarrow (i, j, \text{ctr}, m, c)$	require $\text{pendingToken}[i, j, \text{ctr}] \neq \perp$
If $\text{entry} \notin S$ then	$\text{verTok} \leftarrow \text{pendingToken}[i, j, \text{ctr}]$
$\text{bad}_0 \leftarrow \text{true}$ // $\mathcal{B}_{\text{INT-PCTXT-SA}}$ wins.	$(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence}')$
$\text{abort}(\text{false})$ // $\mathcal{G}_{[1, \infty)}$	If $\text{acc} = \text{false}$ then return $(\mathcal{G}_j^l, \text{false})$
Else $S \leftarrow S \setminus \{\text{entry}\}$	If $\text{evidence}' \neq \text{sendEvidence}[i, \text{ctr}]$ then
$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \text{verTok}$	$\text{bad}_0 \leftarrow \text{true}$ // $\mathcal{B}_{\text{INT-PCTXT-SA}}$ wins.
$\mathcal{G} \leftarrow \text{AddRecvVrtx}(\mathcal{G}, i, j, \text{ctr})$	$\text{abort}(\text{false})$ // $\mathcal{G}_{[1, \infty)}$
If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then	$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \perp$
$\text{bad}_1 \leftarrow \text{true}$ // $\mathcal{B}_{\text{CORR-G-SA}}$ wins.	$\mathcal{G} \leftarrow \text{AddEvidenceReceipt}(\mathcal{G}, i, j, \text{ctr})$
$\text{win} \leftarrow \text{true}$ // $\mathcal{G}_{[0, 2)}$	If $\text{redact}(\mathcal{G}_j^l) \neq \text{restrict}(\text{redact}(\mathcal{G}), j)$ then
Return $(\mathcal{G}_j^l, m, \text{ctr})$	$\text{bad}_1 \leftarrow \text{true}$ // $\mathcal{B}_{\text{CORR-G-SA}}$ wins.
	$\text{win} \leftarrow \text{true}$ // $\mathcal{G}_{[0, 2)}$
	Return $(\mathcal{G}_j^l, \text{acc})$

Fig. 33: Games $\mathcal{G}_0$ – $\mathcal{G}_2$ for the proof of Theorem 9. Code added for game transitions is highlighted in green.

Adversary $\mathcal{B}_{\text{INT-PCTXT-SA}}^{\text{SEND, RECV, VERIFY}}$	$\text{SIMRECV}(i, j, c)$
$\mathcal{A}_{\text{CA-SA}}^{\text{SIMSEND, SIMRECV, SIMVERIFY}}; \text{Return } \perp$	Return $\text{RECV}(i, j, c)$
$\text{SIMSEND}(i, m)$	$\text{SIMVERIFY}(i, j, \text{ctr}, \text{evidence}')$
Return $\text{SEND}(i, m)$	Return $\text{VERIFY}(i, j, \text{ctr}, \text{evidence}')$

Fig. 34: Adversary $\mathcal{B}_{\text{INT-PCTXT-SA}}$ for the proof of Theorem 9. The adversary's calls to the oracles of the server-aided integrity game are highlighted in orange.

conditions that set bad_1 . These conditions are a difference between the receiver's redacted local graph and the restriction of the redacted global graph, and an `APPLYCHKPT` call that reuses a transcript hash.

C Server-aided proofs (Proof of Theorem 9)

Overview. This proof uses the games in Fig. 33. The adversaries $\mathcal{B}_{\text{INT-PCTXT-SA}}$ and $\mathcal{B}_{\text{CORR-G-SA}}$ appear in Fig. 34 and Fig. 35, respectively. We establish the following claims.

- $\text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CA-SA}}(\mathcal{A}_{\text{CA-SA}}) = \Pr[\mathcal{G}_0]$
- $\Pr[\mathcal{G}_0] - \Pr[\mathcal{G}_1] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}]$
- $\Pr[\text{bad}_0^{\mathcal{G}_1}] \leq \text{Adv}_{\text{GC}_{\text{SA}}, n}^{\text{INT-PCTXT-SA}}(\mathcal{B}_{\text{INT-PCTXT-SA}})$
- $\Pr[\mathcal{G}_1] - \Pr[\mathcal{G}_2] \leq \Pr[\text{bad}_1^{\mathcal{G}_2}]$
- $\Pr[\text{bad}_1^{\mathcal{G}_2}] \leq \text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CORR-G-SA}}(\mathcal{B}_{\text{CORR-G-SA}})$
- $\Pr[\mathcal{G}_2] = 0$

Adversary $\mathcal{B}_{\text{CORR-G-SA}}^{\text{SEND,RECV,VERIFY}}((st_1, \mathcal{G}_1^l), \dots, (st_n, \mathcal{G}_n^l))$	$\text{SIMSEND}(i, m)$
$S \leftarrow \emptyset$	$\text{ctr}_i \leftarrow \text{ctr}_i + 1$
$\text{ctr}_j \leftarrow 0 \quad \forall j \in [n]$	$(st_i, \mathcal{G}_i^l, c, \text{evidence}) \leftarrow \text{SEND}(i, m)$
$\mathcal{A}_{\text{CA-SA}}^{\text{SIMSEND, SIMRECV, SIMVERIFY}}$	$\text{sendEvidence}[i, \text{ctr}_i] \leftarrow \text{evidence}$
Return $\perp$	$S \leftarrow S \cup \{(i, j, \text{ctr}_i, m, c) \mid j \in [n] \setminus \{i\}\}$
$\text{SIMRECV}(i, j, c)$	Return $(\mathcal{G}_i^l, c, \text{evidence})$
$(st_j, \mathcal{G}_j^l, m, \text{ctr}, \text{verTok}) \leftarrow \text{GC}_{\text{SA}}.\text{Recv}(j, st_j, \mathcal{G}_j^l, i, c)$	$\text{SIMVERIFY}(i, j, \text{ctr}, \text{evidence}')$
If $m = \perp$ then return $\perp$	require $\text{pendingToken}[i, j, \text{ctr}] \neq \perp$
If $(i, j, \text{ctr}, m, c) \notin S$ then abort ($\perp$)	$\text{verTok} \leftarrow \text{pendingToken}[i, j, \text{ctr}]$
$S \leftarrow S \setminus \{(i, j, \text{ctr}, m, c)\}$	$(st_j, \mathcal{G}_j^l, \text{acc}) \leftarrow \text{GC}_{\text{SA}}.\text{Verify}(j, st_j, \mathcal{G}_j^l, \text{verTok}, \text{evidence}')$
$\text{RECV}(i, j, \text{ctr})$	If $\text{acc} = \text{false}$ then return $(\mathcal{G}_j^l, \text{false})$
$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \text{verTok}$	If $\text{evidence}' \neq \text{sendEvidence}[i, \text{ctr}]$ then abort ($\perp$)
Return $(\mathcal{G}_j^l, m, \text{ctr})$	$\text{VERIFY}(i, j, \text{ctr})$
	$\text{pendingToken}[i, j, \text{ctr}] \leftarrow \perp$
	Return $(\mathcal{G}_j^l, \text{acc})$

Fig. 35: Adversary $\mathcal{B}_{\text{CORR-G-SA}}$ for the proof of [Theorem 9](#). The adversary's calls to the oracles of the graph-correctness game are highlighted in **orange**.

Combining the claims gives

$$\begin{aligned}
& \text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CA-SA}}(\mathcal{A}_{\text{CA-SA}}) \\
&= \Pr[\mathcal{G}_0] \leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\mathcal{G}_1] \\
&\leq \Pr[\text{bad}_0^{\mathcal{G}_1}] + \Pr[\text{bad}_1^{\mathcal{G}_2}] + \Pr[\mathcal{G}_2] \\
&\leq \text{Adv}_{\text{GC}_{\text{SA}}, n}^{\text{INT-PCTXT-SA}}(\mathcal{B}_{\text{INT-PCTXT-SA}}) \\
&\quad + \text{Adv}_{\text{GC}_{\text{SA}}, n, \text{restrict}, \text{redact}}^{\text{CORR-G-SA}}(\mathcal{B}_{\text{CORR-G-SA}}).
\end{aligned}$$

Game $\mathcal{G}_0$. Game $\mathcal{G}_0$ is the CA-SA game augmented by the code highlighted in **green**. The set S stores the entries for intended deliveries of honest sends, and $\text{sendEvidence}[i, \text{ctr}]$ records the evidence object returned by the corresponding honest $\text{GC}_{\text{SA}}.\text{Send}$ call. The added code records these entries and evidence and sets bad_0 and bad_1 . None of the added code changes oracle outputs or win in $\mathcal{G}_0$, establishing the initial-game equality above.

Transition $\mathcal{G}_0 \rightarrow \mathcal{G}_1$. Game $\mathcal{G}_1$ aborts with output **false** under two added conditions. The oracle $\text{RECV}(i, j, c)$ invokes the algorithm $\text{GC}_{\text{SA}}.\text{Recv}$. If the algorithm returns $m \neq \perp$ and $\text{entry} \notin S$, the game aborts. The oracle $\text{VERIFY}(i, j, \text{ctr}, \text{evidence}')$ invokes the algorithm $\text{GC}_{\text{SA}}.\text{Verify}$. If the algorithm returns $\text{acc} = \text{true}$ and $\text{evidence}' \neq \text{sendEvidence}[i, \text{ctr}]$, the game aborts. Games $\mathcal{G}_0$ and $\mathcal{G}_1$ are identical until bad_0 is set, so the Fundamental Lemma establishes the transition bound above. For the associated bad-event bound, $\mathcal{B}_{\text{INT-PCTXT-SA}}$ runs $\mathcal{A}_{\text{CA-SA}}$; see [Fig. 34](#). For each oracle query made by $\mathcal{A}_{\text{CA-SA}}$, $\mathcal{B}_{\text{INT-PCTXT-SA}}$ invokes the matching server-aided integrity oracle and returns its response unchanged. Until bad_0 is set, this reproduces $\mathcal{G}_1$'s oracle interface for $\mathcal{A}_{\text{CA-SA}}$. Whenever $\mathcal{A}_{\text{CA-SA}}$'s RECV or VERIFY query sets bad_0 in $\mathcal{G}_1$, the identically named server-aided integrity call lets $\mathcal{B}_{\text{INT-PCTXT-SA}}$ win in the INT-PCTXT-SA game.

Transition $\mathcal{G}_1 \rightarrow \mathcal{G}_2$. Game $\mathcal{G}_2$ removes the win assignments made when a receiver's redacted local graph differs from the restriction of the redacted global graph for that receiver. Games $\mathcal{G}_1$ and $\mathcal{G}_2$ are identical until bad_1 is set, so the Fundamental Lemma establishes the transition bound above. For the associated bad-event bound, $\mathcal{B}_{\text{CORR-G-SA}}$ runs $\mathcal{A}_{\text{CA-SA}}$; see [Fig. 35](#). $\mathcal{B}_{\text{CORR-G-SA}}$ uses the graph-correctness game's oracles to simulate $\mathcal{G}_2$'s oracle interface for $\mathcal{A}_{\text{CA-SA}}$. To answer $\mathcal{A}_{\text{CA-SA}}$'s RECV query, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMRECV procedure first evaluates the algorithm $\text{GC}_{\text{SA}}.\text{Recv}$ on $\mathcal{B}_{\text{CORR-G-SA}}$'s stored receiver state and local graph. To answer $\mathcal{A}_{\text{CA-SA}}$'s VERIFY query, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMVERIFY procedure first evaluates the algorithm $\text{GC}_{\text{SA}}.\text{Verify}$. If either algorithm accepts, $\mathcal{B}_{\text{CORR-G-SA}}$ first associates the result with the corresponding honest send in $\mathcal{G}_2$. Only then does $\mathcal{B}_{\text{CORR-G-SA}}$ call the corresponding graph-correctness oracle.

By the server-aided syntax, the algorithms $\text{GC}_{\text{SA}}.\text{Recv}$ and $\text{GC}_{\text{SA}}.\text{Verify}$ are deterministic. A Recv rejection with $m = \perp$ preserves the input state and local graph. A Verify rejection with $\text{acc} = \text{false}$ does the same. To answer $\mathcal{A}_{\text{CA-SA}}$'s SEND query, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMSEND procedure calls the oracle SEND of the graph-correctness game. The SIMSEND procedure stores the returned state and local graph, records S and sendEvidence exactly as $\mathcal{G}_2$ does, and returns $(\mathcal{G}_i^l, c, \text{evidence})$ to $\mathcal{A}_{\text{CA-SA}}$. $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMRECV procedure runs the algorithm $\text{GC}_{\text{SA}}.\text{Recv}$ on $\mathcal{B}_{\text{CORR-G-SA}}$'s stored receiver values. If this evaluation produces $m = \perp$, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMRECV returns $\perp$. If the resulting entry is absent from S , $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMRECV aborts with $\perp$. Otherwise, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMRECV removes the matching entry from S and calls the oracle $\text{RECV}(i, j, \text{ctr})$ of the graph-correctness game. The SIMRECV procedure then stores the verification token returned by the earlier evaluation of $\text{GC}_{\text{SA}}.\text{Recv}$ and returns $(\mathcal{G}_j^l, m, \text{ctr})$ to $\mathcal{A}_{\text{CA-SA}}$. $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMVERIFY procedure runs the algorithm $\text{GC}_{\text{SA}}.\text{Verify}$ on $\mathcal{B}_{\text{CORR-G-SA}}$'s stored receiver values. If this evaluation produces $\text{acc} = \text{false}$, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMVERIFY returns $(\mathcal{G}_j^l, \text{false})$. If $\text{evidence}'$ differs from $\text{sendEvidence}[i, \text{ctr}]$, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMVERIFY aborts with $\perp$. Otherwise, $\mathcal{B}_{\text{CORR-G-SA}}$'s SIMVERIFY calls the oracle $\text{VERIFY}(i, j, \text{ctr})$ of the graph-correctness game. On return, SIMVERIFY clears the stored verification token and returns $(\mathcal{G}_j^l, \text{acc})$ to $\mathcal{A}_{\text{CA-SA}}$. On each such oracle call, the graph-correctness game compares the receiver's redacted local graph with the restriction of the redacted global graph. A difference is exactly the condition that sets bad_1 in $\mathcal{G}_2$.

More precisely, when SIMRECV begins, $\mathcal{B}_{\text{CORR-G-SA}}$'s stored $(st_j, \mathcal{G}_j^l)$ equals the corresponding pair maintained by the graph-correctness game for every j . The table pendingToken maintained by $\mathcal{B}_{\text{CORR-G-SA}}$ also agrees with the corresponding table in the graph-correctness game. The same invariant holds when SIMVERIFY begins. The graph-correctness game supplies its initial pairs to $\mathcal{B}_{\text{CORR-G-SA}}$. When $\mathcal{B}_{\text{CORR-G-SA}}$ answers $\mathcal{A}_{\text{CA-SA}}$'s SEND query as described above, $\mathcal{B}_{\text{CORR-G-SA}}$ stores the state and local graph returned by the oracle SEND of the graph-correctness game. If either algorithm $\text{GC}_{\text{SA}}.\text{Recv}$ or $\text{GC}_{\text{SA}}.\text{Verify}$ rejects, $\mathcal{B}_{\text{CORR-G-SA}}$'s stored state and local graph remain unchanged. In that case, $\mathcal{B}_{\text{CORR-G-SA}}$ makes no graph-correctness-oracle call. A missing entry or evidence mismatch terminates $\mathcal{B}_{\text{CORR-G-SA}}$.

Consider any RECV or VERIFY call that $\mathcal{B}_{\text{CORR-G-SA}}$ makes to the graph-correctness game. The invoked oracle repeats $\mathcal{B}_{\text{CORR-G-SA}}$'s preceding deterministic algorithm evaluation from the equal pre-query state. For a receive, the entry in S supplies the same ciphertext and send counter. For a verification, $\mathcal{B}_{\text{CORR-G-SA}}$ first checks $\text{evidence}' = \text{sendEvidence}[i, \text{ctr}]$. Only if this equality holds does $\mathcal{B}_{\text{CORR-G-SA}}$ call the graph-correctness oracle VERIFY . That oracle evaluates the algorithm $\text{GC}_{\text{SA}}.\text{Verify}$ with the evidence used in $\mathcal{B}_{\text{CORR-G-SA}}$'s first evaluation of that algorithm. Thus the resulting state pairs are equal, and both copies of pendingToken undergo the same create-or-consume update.

Consequently, each call to the graph-correctness oracle RECV or VERIFY uses the same send-counter value as the corresponding branch of $\mathcal{G}_2$. The invoked oracle runs the corresponding receive-side algorithm from the same pre-query receiver state st_j and local graph $\mathcal{G}_j^l$. In the graph, that counter value serves as the send-event identifier. Thus, any accepted receive or verification for which these two resulting graph views differ sets bad_1 . On that event, $\mathcal{B}_{\text{CORR-G-SA}}$ wins in the CORR-G-SA game.

Game $\mathcal{G}_2$. Game $\mathcal{G}_2$ contains no active win condition, establishing the final claim above.

D Related work

Ordering in distributed systems. Causal ordering has long been studied in distributed systems. The partial order imposed by our causality graphs is Lamport's happened-before relation [Lam78]; however, we derive it using causality graphs rather than logical clocks. Early work on causal broadcast [BSS91] builds on Lamport's logical clocks. In these systems, each party keeps a vector clock with one counter per participant, recording the number of that participant's messages the party has accepted so far. A sender attaches its vector clock to each message it sends, so a receiver can compare the sender's clock with its own and determine whether it has already seen all messages that causally precede the received message. If an earlier message is still missing, the receiver holds the new message until that predecessor arrives. Causally consistent data stores [LFKA11, DIRZ14, ASB18] provide the same guarantees, but use different techniques (explicit dependency lists [LFKA11], a single physical timestamp [DIRZ14], and mergeable deltas [ASB18]). All these systems assume honest nodes and reliable (or eventual) delivery, so a receiver

can trust the ordering data on each message. However, in messaging, the delivery service is untrusted, so the protocol must guarantee integrity of ordering data and be resilient to dropped or reordered messages. In settings with malicious participants, prior work [RG93, RB94] shows that achieving the same guarantees is expensive, requiring either blocking delivery or adding rounds of communication. Although that setting differs from ours, achieving causal awareness in group messaging against malicious insiders would likely incur similar costs.

Ordering in two-party messaging. Barooti et al. [BCC⁺23] study the detection of active attacks under state compromise in the two-party setting and give a lower bound: any scheme achieving their $r - \text{RID}$ notion must include the entire conversation history (i.e., every prior send and receive) with each message, so ciphertext size grows with the length of the conversation. Their goal is different from and stronger than ours, so the bound does not apply to causal awareness.

Chen and Fischlin [CF24] extend Lamport’s happened-before relation to a two-party causality graph and define causality preservation by requiring a party’s locally reconstructed causality graph to equal the global graph restricted to that party’s view. In other words, a receiver should be able to reconstruct all events that causally preceded its most recent receive action. They show that Signal does not achieve their notion and give a generic compiler that turns any bidirectional channel into a causality-preserving one. Extending Chen and Fischlin’s causality preservation notion to the group setting is nontrivial. The strongest variant (rGMax) of our causal awareness notion extends their notion to the group setting. We show that achieving this notion is impractical and suggest weaker, more efficient variants.

Chen and Fischlin also study causality for two-party message franking [CF24]. Namavari and Ristenpart [NR25] extend this work to transcript franking, i.e., reports that contain several messages and preserve their causal order, in a setting where the platform is trusted for ordering but clients may be malicious.

Ordering in group messaging. Unger et al. [UDB⁺15] informally define three ordering goals for secure messaging: *speaker consistency*, which guarantees that all users agree on the sequence of messages sent by each user; *causality preservation*, which guarantees that a message is shown only after the messages that causally precede it; and *global transcript*, which requires all users to see all messages in the same (total) order. Causal awareness corresponds to none of these goals. It neither requires delaying the display of a message until its predecessors arrive nor requires a total order over all messages. Instead, it requires that, after a user accepts a message, the user can reconstruct its causal context.

Goldberg et al. [GUVC09] informally define a consensus notion under which users who compare their transcripts pairwise at the end of a chat must have the same set of messages with the same authors. Their consensus notion does not provide any ordering guarantees and is achieved only after the chat ends. Signal’s early group-chat design [Mar14] sketched a per-message ancestor-hash DAG to achieve Goldberg et al.’s consistency notion [GUVC09]. Mobile Conversation With Privacy and Integrity (CoWPI) [SH19] requires a total order over all messages, but each message must wait for approval from its order-enforcing services, at least one of which must be honest.

Marson [Mar17] defines communication graphs and channels that record a user’s causal history and require that record to match the true order of events. Eugster, Marson, and Poettering [EMP18] build causal broadcast channels that delay delivery of a message until its predecessors arrive. Neither considers message loss or out-of-order delivery.

The Matrix Event Graph [JBHH21] is a server-side DAG that links each event to its parent events and is replicated across (room) servers. However, its architecture assumes honest servers, providing no way for a client to authenticate the graph itself.

SYM-GOTR [SVH18] is a synchronous group-messaging protocol whose members run constant-size checks to confirm that they share the same transcript and otherwise detect deviations. Liu-Zhang, Portmann, and Rito [LZPR24] formalize a consistency notion in the malicious insider setting. In their construction, each message carries its causal predecessors, and a receiver accepts a message only after it has received all its predecessors. Collectively, these works rely on assumptions we do not make: they trust an ordering service [JBHH21], assume synchronicity [SVH18], or block messages until their predecessors arrive [LZPR24].

Ordering in CGKA and MLS. The Messaging Layer Security (MLS) standard [BBR⁺23,BRO⁺25] defines a transcript consistency notion that requires all participants in a group chat to share the same totally ordered transcript. However, it requires transcript consistency only for *handshake* messages, i.e., messages that evolve the cryptographic state of the group (when parties are added or removed, or a party updates their keys), not for application messages. Related work studies concurrent group-state changes [AAN⁺22,AMT23] and decentralized variants that operate without a central ordering service but assume causal broadcast, allowing missing predecessors to be fetched first [WKHB21,AAN⁺24]. The latter works address forks in group state in the style of fork-linearizable storage [CsS07]. None of these works addresses ordering guarantees for application messages, which are our focus.

Ordering in message franking. Message franking lets the recipient of an abusive message report it to the platform, which can then verify that the reported message was really sent without having seen the plaintext at delivery time. Message franking assumes a trusted server but allows potentially malicious participants. Chen and Fischlin [CF24] add causality to two-party message franking, so a report attests not only to a message’s content but also to its causal position. Namavari and Ristenpart [NR25] extend this guarantee from single messages to multi-message transcripts.

Attacks against ordering in deployed messengers. Schliep, Kariniemi, and Hopper [SKH17] show that, in earlier implementations of Signal’s two-party and group protocols, a compromised server could quietly reorder and drop messages and alter chat membership. Jaeger and Kumar [JK25] show reordering attacks against MLS and Session in the malicious insider setting. Rösler et al. [RMS18] find that a malicious server can present different subsets or orderings of messages to different users in Signal, WhatsApp, and Threema without the users being able to detect it. Paterson et al. [PST23] and Albrecht et al. [AMPS26] show that even per-sender ordering is not guaranteed in Threema and Telegram, respectively. Gegenhuber et al. [GGG⁺26] show that malicious members can force different transcripts on honest users in Signal, WhatsApp, Threema, and iMessage by exploiting the applications’ fallback behavior after a decryption failure or sending distinct sender keys to different receivers.